AUTOMATIC SYNTHETIC BENCHMARK GENERATION FOR MULTICORE SYSTEMS

by

Etem Deniz

B.S., Computer Engineering, Dokuz Eylül University, 2009

M.Sc., Computer Engineering, Boğaziçi University, 2011

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

Graduate Program in Computer Engineering

Boğaziçi University

2015

AUTOMATIC SYNTHETIC BENCHMARK GENERATION FOR MULTICORE SYSTEMS

APPROVED BY:

Assoc. Prof. Alper Şen (Thesis Supervisor)	.....
Prof. Oğuz Tosun	.....
Prof. Haluk Rahmi Topçuoğlu	.....
Prof. Can Özturan	.....
Assoc. Prof. Zeki Bozkuş	.....

DATE OF APPROVAL: 11.12.2015

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Alper Şen for his invaluable guidance, great patience, friendly attitude, and encouraging support. This thesis would not be possible without his efforts.

I would like to thank my thesis committee members Professor Oğuz Tosun, Haluk Rahmi Topçuoğlu, Can Özturan, and Zeki Bozkuş for their valuable efforts and insightful comments.

I am thankful to Jim Holt, Brian Kahne, Michele Reese, and Jaksa Djordjevic from Freescale for providing invaluable comments and feedback and for helping me with hardware measurements.

I would like to acknowledge fellowships and grants that supported my research during my PhD: Semiconductor Research Corporation under task 2082.001, European Commission, Boğaziçi University Research Fund, Turkish Academy of Sciences, and Scientific and Technical Research Council of Turkey (TÜBİTAK).

Last but not least, I would like to dedicate this thesis to my wife Merve and daughters Miray and Melis, for their love, patience, sacrifices, and understanding - they allowed me to spend most of the time on this thesis.

ABSTRACT

AUTOMATIC SYNTHETIC BENCHMARK GENERATION FOR MULTICORE SYSTEMS

We present a novel automated multicore benchmark synthesis framework for multicore systems including CPUs and GPUs with characterization and generation components to speed up architectural simulation of modern architectures. We first identify a set of important application characteristics for CPUs and GPUs. Then, our framework captures these characteristics of original multicore applications and generates synthetic multicore benchmarks from those applications where synthetic benchmarks are a miniaturized form of applications that allow high simulation speeds and act as proxies of proprietary applications. We use parallel software architectural patterns in capturing important characteristics of CPU applications where we apply different machine learning techniques in a novel approach to automatically detect parallel patterns used in applications. In addition, we compare these techniques in terms of accuracy and speed and demonstrate that detecting parallel patterns is crucial for performance improvements and enables many architectural optimizations. The resulting synthetic benchmarks are small, fast, portable, human-readable, and they accurately reflect the key characteristics of the original multicore applications. Our synthetic CPU benchmarks use either Pthreads or Multicore Association (message passing and resource management) libraries and synthetic GPU benchmarks use OpenCL library. To the best of our knowledge, this is the first time synthetic OpenCL benchmarks for GPUs are generated from existing applications. We implement our techniques for CPUs in the MINIME tool and generate synthetic benchmarks. Similarly, we implement our techniques for GPUs in the MINIME-GPU tool and experimentally validate them.

ÖZET

ÇOK ÇEKİRDEKLİ SİSTEMLER İÇİN OTOMATİK KARŞILAŞTIRMA TESTİ YARATMA

Modern çok çekirdekli CPU ve GPU mimari simülasyonunu hızlandırmak için sentetik karşılaştırma uygulamalarını otomatik şekilde yaratan, karakterizasyon ve sentezleme bileşenleri olan yeni sentezleme araçları geliştirdik. İlk olarak, CPU ve GPU sistemleri için önemli olan karakteristiklerin kümesi belirledik. Daha sonra geliştirdiğimiz araçlar ile bu karakteristikleri mevcut uygulamalardan toplayarak bu uygulamaların yüksek hızda simülasyonuna olanak sağlayan minyatür halleri olan sentetik karşılaştırma uygulamaları oluştururduk. CPU uygulamalarının önemli karakteristiklerini toplamak için yazılım mimari kalıplarını kullandık ve çeşitli makine öğrenme tekniklerini uygulayarak bu yazılım mimari kalıpları otomatik olarak tanımladık. Bununla beraber bu makine öğrenme tekniklerini doğruluk ve hız açısından karşılaştırdık ve yazılım mimari kalıplarının tanımlanmasının performans ve mimari eniyileme açısından önemli olduğunu gösterdik. Sentezlenen karşılaştırma uygulamalarımız küçük, hızlı, taşınabilir ve okunabilir olduğu gibi sentezlendikleri gerçek uygulamanın karakteristiklerini de doğru şekilde taklit etmektedir. Sentetik CPU karşılaştırma uygulamalarımız Pthreads veya Multicore Association (mesaj iletim veya kaynak yönetim) kütüphanelerini ve sentetik GPU karşılaştırma uygulamalarımız OpenCL kütüphanesini kullanabilmektedir. Bu çalışma ile varolan GPU uygulamarından ilk kez sentetik OpenCL karşılaştırma uygulaması geliştirildi. CPU tekniklerimiz için MIMIME aracını geliştirdik ve sentetik karşılaştırma uygulamaları yarattık. Benzer şekilde, GPU tekniklerimiz için MINIME-GPU aracını geliştirerek deneylerle tekniklerimizi doğruladık.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . xi
LIST OF TABLES . . . xviii
LIST OF SYMBOLS . . . xx
LIST OF ACRONYMS/ABBREVIATIONS . . . xxiii
1. INTRODUCTION . . . 1
1.1. Contributions . . . 4
1.2. Organization . . . 7
2. BACKGROUND . . . 8
2.1. Software Architectural Patterns . . . 8
2.2. Multicore Programming APIs . . . 10
2.3. GPUs and Programming Models . . . 12
2.3.1. OpenCL Programming Model . . . 12
2.3.2. Target GPU Architecture . . . 14
2.4. Machine Learning Techniques . . . 15
2.4.1. k-Nearest Neighbor . . . 15
2.4.2. Decision Trees . . . 15
2.4.3. Naive Bayes Classifier . . . 16
2.4.4. Neural Networks . . . 16
2.4.5. Principal Component Analysis with K-means . . . 17
2.5. Benchmarks for Performance Evaluation . . . 18
2.5.1. Synthetic Benchmarks . . . 19
3. SYNTHETIC BENCHMARK GENERATION FOR MULTICORE CPUs . . . 20
3.1. Overview . . . 20
3.2. High-level Framework . . . 22
3.3. Multicore Benchmark Characterization . . . 23
3.3.1. Data Sharing Characteristics . . . 23

3.3.2. Thread Communication Characteristics 25
3.3.3. General Threading Characteristics 25
3.3.4. Performance Characteristics 26
3.3.5. Characterization Tools 27
3.4. Parallel Pattern Recognition 27
3.5. Pattern-Aware Synthetic Benchmark Generation 30
3.5.1. Code Generation for High Level Metrics 30
3.5.2. Similarity Measurement 34
3.5.3. Code Generation for Low Level Metrics 34
3.5.4. A Detailed Example of CPU Benchmark Synthesis 36
3.6. Synthetic Benchmark Generation for Embedded Multicore Systems . . 37
3.7. Experiments 41
3.7.1. Evaluation of Benchmark Synthesis 42
3.7.2. Assessing Similarity 45
3.7.3. Assessing Architecture Changes 47
3.7.4. Assessing Input Changes 48
3.7.5. Correlation between Parallel Pattern Score and Overall Similarity Score 49
3.7.6. Synthetic Benchmark Generation for Embedded Multicore Systems 51
3.7.7. Discussion 56
3.8. Summary 64
4. THREAD-LEVEL SYNTHETIC BENCHMARKS FOR MULTICORE CPUs 65
4.1. Overview 65
4.2. Thread-level Synthetic Benchmark Development Framework 66
4.2.1. Benchmark Characterizer 67
4.2.2. Benchmark Generator 68
4.3. Application-level versus Thread-level Synthetic Benchmarks 70
4.4. Experiments 72
4.4.1. Decision tree based parallel pattern recognition 77
4.5. Summary 78
5. SYNTHETIC BENCHMARK GENERATION FOR GPUs 79

5.1. Overview . . . 79
5.2. High-level Framework . . . 81
5.3. Benchmark Characterization . . . 82
5.4. Benchmark Generation . . . 86
5.4.1. Similarity Measurement . . . 87
5.4.2. Code (Block) Generation . . . 87
5.4.3. A Detailed Example of GPU Benchmark Synthesis . . . 92
5.5. Experiments . . . 93
5.5.1. Simulation and Benchmarks . . . 94
5.5.2. Applying PCA to Validate the Importance of Characteristics . . . 95
5.5.3. Synthetic Benchmark Generation Results . . . 96
5.5.4. Assessing Similarity . . . 98
5.5.5. Validation of Synthetic Benchmarks on Real Hardware . . . 99
5.5.6. Assessing Architecture Changes . . . 102
5.5.6.1. Synthesis for GPUs having Derived Architectural Configurations . . . 105
5.5.7. Assessing Input Changes . . . 110
5.6. Discussion . . . 111
5.7. Summary . . . 113
6. USING MACHINE LEARNING TECHNIQUES TO DETECT PARALLEL PATTERNS OF MULTI-THREADED APPLICATIONS . . . 114
6.1. Overview . . . 114
6.1.1. Motivation for Classifying Parallel Patterns . . . 116
6.2. Parallel Pattern Classification Using Machine Learning . . . 118
6.3. Characterization of Multi-threaded Applications . . . 118
6.3.1. Data Preparation . . . 120
6.3.1.1. Data Collection . . . 120
6.3.1.2. Data Pre-processing . . . 121
6.4. Experiments . . . 123
6.4.1. Pattern Classification Results . . . 124
6.4.2. Decision Trees . . . 126

6.4.2.1. Construction of a Decision Tree . . . 126
6.4.2.2. Using the Decision Tree . . . 128
6.4.3. Naive Bayes Classifier . . . 129
6.4.3.1. Construction of a Naive Bayes Classifier . . . 129
6.4.3.2. Using the Naive Bayes Classifier . . . 131
6.4.4. Neural Networks . . . 131
6.4.4.1. Construction of a Neural Network . . . 131
6.4.4.2. Using the Neural Network . . . 133
6.4.5. Principal Component Analysis with K-means . . . 133
6.4.5.1. Construction of a Principal Component Analysis with K-means . . . 133
6.4.5.2. Using the Principal Component Analysis with K-means 134
6.4.6. Feature Selection for Our Machine Learning Techniques . . . 136
6.4.7. Comparison of Our Machine Learning Techniques . . . 138
6.4.8. Using Parallel Patterns Detection Results . . . 139
6.4.8.1. Synthetic Benchmark Generation . . . 139
6.4.8.2. Correlation of Parallel Pattern with Synthetic Benchmark Similarity . . . 141
6.5. Discussion . . . 142
6.5.1. Data Collection Scalability . . . 142
6.5.2. Training Set Size and Cross Validation . . . 143
6.5.3. Impact of Multiple Inputs . . . 144
6.5.4. Multiple Parallel Patterns . . . 145
6.6. Summary . . . 145
7. RELATED WORK . . . 147
7.1. Synthetic Benchmark Generation for Multicore CPUs . . . 147
7.1.1. Software Architectural Patterns . . . 147
7.1.2. Benchmark Characterization . . . 147
7.1.3. Synthetic Benchmark Generation . . . 148
7.2. Thread-level Synthetic Benchmark for CPUs . . . 150
7.2.1. Benchmark Characterization . . . 150

7.3. Synthetic Benchmark Generation for GPUs 151
7.3.1. Benchmark Characterization 151
7.3.2. Synthetic Benchmark Generation 152
7.4. Machine Learning Techniques to Detect Parallel Patterns 153
7.4.1. Parallel Pattern Detection 153
7.4.2. Machine Learning Techniques 154
8. CONCLUSIONS AND FUTURE WORK 157
8.1. Summary 157
8.2. Future Work 161
REFERENCES 164

LIST OF FIGURES

Figure 2.1. Parallel patterns for software. 9

Figure 2.2. OpenCL platform model. 13

Figure 2.3. OpenCL programming and memory model. 13

Figure 2.4. Simplified AMD Southern Islands GPU architecture [1]. 14

Figure 3.1. MINIME: Pattern-aware multicore benchmark synthesizer architecture. 22

Figure 3.2. Code block to increment/decrement IPC, CMR, and BMR. 35

Figure 3.3. Parallel patterns scores of matrix multiplication. 37

Figure 3.4. Synthetic matrix multiplication benchmark. 38

Figure 3.5. Comparison of IPC between the synthetic and original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III. 45

Figure 3.6. Comparison of CMR between the synthetic and the original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III. 46

Figure 3.7. Comparison of BMR between the synthetic and the original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III. 46

Figure 3.8. IPC values of original benchmarks for small, medium, and large inputs on System-I. 48

Figure 3.9. Linear regression analysis between Data Sharing Score + Thread Communication Score and Total Pattern Score. 49

Figure 3.10. Linear regression analysis of `Bodytrack` for parallel pattern and overall similarity score relation. 50

Figure 3.11. Linear regression analysis of `ippktcheck` for parallel pattern and overall similarity score relation. 50

Figure 3.12. Overall similarity scores of synthetic benchmarks from PARSEC for MRAPI/MCAPI on HW1. 53

Figure 3.13. Overall similarity scores of synthetic benchmarks from Rodinia for MRAPI/MCAPI on HW1. 54

Figure 3.14. Thread Communication scores of the synthetic benchmarks from PARSEC for MRAPI and MCAPI. 55

Figure 3.15. Thread Communication scores of the synthetic benchmarks from Rodinia for MRAPI and MCAPI. 55

Figure 3.16. Comparison of CCR between the synthetic and the original benchmarks from PARSEC. 57

Figure 3.17. Comparison of CCR between the synthetic and the original benchmarks from Rodinia. 57

Figure 3.18. Comparison of IPC between the synthetic and the original benchmarks from PARSEC for MRAPI. 58

Figure 3.19. Comparison of IPC between the synthetic and the original benchmarks from Rodinia for MRAPI. 58

Figure 3.20. Comparison of CMR between the synthetic and the original benchmarks from PARSEC. 59

Figure 3.21. Comparison of CMR between the synthetic and the original benchmarks from Rodinia. 59

Figure 3.22. Comparison of BMR between the synthetic and the original benchmarks from PARSEC. 60

Figure 3.23. Comparison of BMR between the synthetic and the original benchmarks from Rodinia. 60

Figure 3.24. Overall similarity scores of synthetic benchmarks from PARSEC for MRAPI/MCAPI on HW2. 61

Figure 3.25. Overall similarity scores of synthetic benchmarks from Rodinia for MRAPI/MCAPI on HW2. 61

Figure 3.26. Comparison of IPC between the synthetic and the original benchmarks from PARSEC on HW2. 62

Figure 3.27. Comparison of IPC between the synthetic and the original benchmarks from Rodinia on HW2. 62

Figure 4.1. MINIME (thread-level): multi-threaded benchmark development framework. 67

Figure 4.2. Code block to increment/decrement IPC. 69

Figure 4.3. Characteristics of `Blackscholes` benchmark and its application-level synthetic. 70

Figure 4.4. Characteristics of `Blackscholes` benchmark and its thread-level synthetic. 70

Figure 4.5. Characteristics of `Blackscholes` benchmark and thread-level synthetic of Blackscholes with 8 threads. 75

Figure 4.6. Average, maximum, and minimum thread similarity scores of all thread-level synthetic benchmarks. 76

Figure 4.7. Comparison of average thread similarity scores for application-level and thread-level synthetic of all benchmarks. 77

Figure 5.1. MINIME-GPU: multicore benchmark synthesizer for GPUs. 81

Figure 5.2. Host program of a synthetic benchmark. 85

Figure 5.3. Kernel program of a synthetic benchmark. 85

Figure 5.4. Sample code blocks to increment/decrement the values of kernel program (instruction throughput and computation-to-memory access) characteristics. 88

Figure 5.5. Sample code blocks to increment/decrement the values of kernel program (dynamic memory instruction mix, memory efficiency, and compute unit occupancy) characteristics. 89

Figure 5.6. Synthetic benchmark (kernel program) for `QuasiRandomSequence`. 92

Figure 5.7. Principal components and their variance. 95

Figure 5.8. PCA loadings with respect to each characteristic. 96

Figure 5.9. Comparison of IPC between the synthetic and original benchmarks. 97

Figure 5.10. Comparison of memory coalescing between the synthetic and original benchmarks. 99

Figure 5.11. Comparison of VALU utilization between the synthetic and original benchmarks on real hardware (HD 7950). 102

Figure 5.12. Comparison of SGPRs utilization between the synthetic and original benchmarks on real hardware (HD 7950). 102

Figure 5.13. Comparison of sensitivity to architecture changes for `BitonicSort` and `QuasiRandomSequence`. 102

Figure 5.14. The IPC error score for different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770). 103

Figure 5.15. The hit ratio error score for different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770). 103

Figure 5.16. IPCs of original and synthetic benchmarks on different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770). 104

Figure 5.17. Hit ratios of original and synthetic benchmarks on different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770). 104

Figure 5.18. The number of (in-flight) wavefronts error score for the GPUs having derived architectural configurations (HD 7970, HD 7870d, HD 7850d, and HD 7770d). 105

Figure 5.19. Number of (in-flight) wavefronts of original and synthetic benchmarks on the GPUs having derived architectural configurations (HD 7970, HD 7870d, HD 7850d, and HD 7770d). 105

Figure 5.20. The IPC error score for the GPUs having derived architectural parameters (HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3). . 107

Figure 5.21. IPCs of original and synthetic benchmarks on the GPUs having derived architectural parameters (HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3). 107

Figure 5.22. The cache hit ratio error score for different cache configurations (*Config-0*, *Config-1*, *Config-2* and *Config-3*). 109

Figure 5.23. Cache hit ratios of original and synthetic benchmarks on different cache configurations (*Config-0*, *Config-1*, *Config-2* and *Config-3*). . 109

Figure 5.24. The IPC values of original benchmarks for small, medium, and large inputs on HD 7970. 110

Figure 6.1. Decision tree for parallel pattern classification. 127

Figure 6.2. ROC curve of our decision tree. 128

Figure 6.3. Confusion matrix of the naive Bayes classifier using the Gaussian method on the training set. 129

Figure 6.4. Confusion matrix of the naive Bayes classifier using the Kernel Density Estimation method on the training set. 130

Figure 6.5. Configuration of our neural network. 131

Figure 6.6. Training performance of the neural network. 132

Figure 6.7. Confusion matrix of the neural network on the training set. 133

Figure 6.8. Principal components and their variance. 134

Figure 6.9. The PCA space that is projected into 2 dimensions (PC1 and PC2). 135

Figure 6.10. Accuracies of our machine learning techniques using the selected sub-characteristics proposed by MI. 136

Figure 6.11. Characterization speedup of decision tree over kNN in Chapter 3. 138

LIST OF TABLES

Table 1.1. Our synthetic benchmarks. 6

Table 3.1. CPU benchmark characteristics. 24

Table 3.2. Data sharing reference behavior [2]. 29

Table 3.3. Thread communication reference behavior. 29

Table 3.4. General threading reference behavior. 29

Table 3.5. Algorithm to generate the code for a thread t_i based on parallel pattern. 30

Table 3.6. Multicore machine configurations. 41

Table 3.7. Pattern recognition and synthesis results. 43

Table 3.8. Multicore hardware configurations. 51

Table 3.9. Benchmark synthesis results for embedded multicore systems. . . . 52

Table 4.1. Multicore machine configuration. 72

Table 4.2. Benchmark characteristics and pattern classification results. 73

Table 4.3. Thread-level synthetic benchmark generation results. 74

Table 5.1. GPU benchmark characteristics. 83

Table 5.2. GPU architectural configurations. 94

Table 5.3. Synthesis results on AMD HD 7970 platform. 98

Table 5.4. GPU architectural parameters. 107

Table 5.5. GPU cache configurations. 109

Table 6.1. Characteristics of multi-threaded applications. 119

Table 6.2. Pattern classification results. 125

Table 6.3. Cut-offs of our ROC curve. 129

Table 6.4. Comparison of machine learning techniques. 136

Table 6.5. Pattern recognition and synthesis results. 140

Table 8.1. Our synthetic benchmarks with experimental details. 160

LIST OF SYMBOLS

$\#$	number sign, for example '#Cores' indicates 'number of cores'
$\#reader$	the unique number of threads that read the same cacheline
$\#st$	the number of threads that do not have the sub-characteristic x in the 2S range around M
$\#writers$	the unique number of threads that write the same cacheline
$\#wt$	the number of all worker threads
$atss$	average thread similarity score
CC	the number of threads that are communicating in both the original and the synthetic workloads
$chorg$	given a similarity characteristic, ch, the value of ch for the original application
$chsyn$	given a similarity characteristic, ch, the value of ch for the synthetic application
CN	the number of threads that are communicating in the original but not in the synthetic workload
$commTH$	pairwise communicating threads
CV	the coefficient of variation
$errorrate_{mt}$	the error rate for a similarity metric mt
$errorrate_{ch}$	the error rate for a similarity characteristic ch
iss	individual similarity score
iss_{BMR}	individual similarity score for BMR
iss_{CCR}	individual similarity score for CCR
iss_{CMR}	individual similarity score for CMR
iss_{IPC}	individual similarity score for IPC
M	mean
mt	a similarity metric
$mtorg$	given a similarity metric, mt, the value of mt for the original workload
$mtsyn$	given a similarity metric, mt, the value of mt for the synthetic workload

$norm_{PC}$	the normalized program counter sub-characteristic
$norm_x$	the normalized sub-characteristic for sub-characteristic x
$numTH$	the total number of threads during execution
$mtorg$	given a similarity metric, mt, the value of mt for the original application
$mtsyn$	given a similarity metric, mt, the value of mt for the synthetic application
NC	the number of threads that are communicating in the synthetic but not in the original workload
NN	the number of threads that are not communicating in both the original and the synthetic workloads
org_BMR	BMR of the original benchmark
org_CMR	CMR of the original benchmark
org_IPC	IPC of the original benchmark
oss	overall similarity score
oss_{app}	overall similarity score for application-level CPU benchmarks
oss_{cpu}	overall similarity score for CPU benchmarks
oss_{emb}	overall similarity score for embedded CPU benchmarks
oss_{gpu}	overall similarity score for GPU benchmarks
S	standard deviation
$sharedCL$	the ratio of cachelines used for communication to all cachelines used during execution gives shared cachelines
sim_BMR	BMR similarity
sim_CMR	CMR similarity
sim_IPC	IPC similarity
sim_IPC_i	the IPC similarity of thread i
$similarity$	the similarity rate
$sscore_{CMR}$	the similarity score for a similarity metric CMR
$sscore_{IPC}$	the similarity score for a similarity metric IPC
$sscore_{mt}$	the similarity score for a similarity metric mt
$sscore_{TC}$	the similarity score for a similarity metric TC
syn_BMR	BMR of the synthetic benchmark

syn_CMR	CMR of the synthetic benchmark
syn_IPC	IPC of the synthetic benchmark
t_i	i-th thread
tss_i	thread similarity score for a thread i in a benchmark
U	the number of unique start (entry point) program counters

LIST OF ACRONYMS/ABBREVIATIONS

ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
BIC	Bayesian Information Criterion
BMR	Branch Misprediction Rate
CB	Code Block
CCR	Communication to Computation Ratio
CMAR	Computation-to-Memory Access Ratio
CMR	Cache Miss Rate
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DaC	Divide and Conquer
DiscoPoP	Discovery of Potential Parallelism
DS	Data Sharing
DSP	Digital Signal Processor
DT	Decision Tree
EbC	Event-based Coordination
EEMBC	Embedded Microprocessor Benchmark Consortium
FIFO	First In First Out
FIR	Finite Impulse Response
FPR	False Positive Rate
FS	Feature Selection
GCC	GNU Compiler Collection
GD	Geometric Decomposition
GNU	GNU's Not Unix
GPGPU	General Purpose Computing on Graphical Processing Unit

GPU	Graphics Processor Unit
GT	General Threading
IC	Dynamic instruction count
ID3	Iterative Dichotomiser 3
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
KB	Kilobyte
KDE	Kernel Density Estimation
kNN	k-Nearest Neighbor
LLC	Last Level Cache
LLVM	Low Level Virtual Machine
LOC	Lines of Code
LT	Lifetime
MATLAB	MATrix LABoratory
MB	Megabyte
MCA	Multicore Association
MCAPI	Multicore Communication API
MI	Mutual Information
MPI	Message Passing Interface
MRAPI	Multicore Resource Management API
MSE	Mean Square Error
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
NBC	Naive Bayes Classifier
NNW	Neural Network
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OSS	Overall Similarity Score
PAPI	Performance Application Programming Interface

PARSEC	Princeton Application Repository for Shared-Memory Computers
PC	Program Counter
PCA	Principal Component Analysis
PCs	Principal Components
PL	Parallel Pattern type
Pl	Pipeline
POSIX	Portable Operating System Interface for Unix
RCT	Ratio of Communicating Threads
RCV	Ratio of Communication Volume
RD	Recursive Data
ROC	Receiver Operating Characteristic
SDK	Software Development Kit
SGPRs	Scalar General-Purpose Registers
SI	Southern Islands
SIMD	Single Instruction Multiple Data
SMP	Symmetrical Multicore Processor
TC	Thread Communication
TLB	Translation Lookaside Buffer
TP	Task Parallel
TPR	True Positive Rate
VALU	Vector Arithmetic Logic Unit

1. INTRODUCTION

The exponential improvement in single core CPU performance has recently come to an end due to thermal and power problems. This has led to the development of multicore CPUs, which can improve performance given parallel applications. Meanwhile, GPUs have become an important platform for data parallel applications thanks to their high parallel architecture. It is important to analyze and optimize these multicore systems to achieve high performance with low power consumption as these systems are becoming very widely used in high performance computing and demanding consumer applications. However, effectively analyzing and optimizing multicore CPUs and GPUs is a nontrivial task that requires domain knowledge in multicore architecture, parallelism fundamentals, and parallel programming paradigms. That is, the task of developing and optimizing multicore systems has been complicated by the concurrent nature of multicore systems since concurrent systems can get into an exponential number of scenarios that cannot be completely analyzed. In addition to the concurrent nature of hardware, concurrent software is also becoming common place where new multicore software paradigms are developed to exploit the performance available in multicore hardware.

Benchmarks are tests of computer systems including CPUs and GPUs that help estimate performance and power consumption of a system on a workload. Computer architects use simulation software (simulators) to model the architecture of the newly developed systems and run benchmarks on these simulators. However, running benchmarks on these simulators are typically many orders of magnitude slower than running them on real machines. Hence, executing many benchmark applications on these simulators will not be possible during the allowed design time due to faster time-to-market necessitates.

One of the commonly used techniques to boost the performance of simulations is to use synthetic benchmarks where synthetic benchmarks are a miniaturized form of benchmarks. These benchmarks can either be derived from existing applications or be

generated from scratch by varying various program characteristics. Synthetic benchmarks do not perform any useful computation, yet they can approximate characteristics of real-life applications, hence they need to be accurate. A synthetic benchmark also needs to be smaller and faster than the original application that it is derived from so that it simulates faster.

The emergence of multicore systems has made parallel benchmark suites ubiquitous. These new generation of benchmarks are used for early design exploration and allow evaluating performance, power consumption, and reliability of the future parallel computer architectures such as multicores, many-cores, accelerators, and supercomputers. Some of the commonly used parallel benchmark suites are PARSEC [3], Rodinia [4], Parboil [5], EEMBC [6], SPLASH-2 [7], and NAS Parallel Benchmarks [8]. These existing benchmark suites are big and rely on presence of shared memory architectures, or Pthreads, OpenMP, and OpenCL libraries as well as uniform CPU Instruction Set Architectures (ISAs). Multicore systems may not be able to use these benchmarks as they may not support such architectures. There is a need for benchmarks suitable for any given infrastructure, that is, SMP or message passing architectures. Furthermore, proprietary customer codes may not be available to the hardware designer.

To meet the preceding needs, we need to develop new benchmarks but benchmark development process is time- and labor-intensive. In this thesis, we present a novel synthetic benchmark generation approach for CPUs and GPUs. Our approach helps developers and researchers focus on analyzing the synthetic benchmark results by hiding the difficulties of the benchmark development from them. Our approach is capable of generating synthetic benchmarks that are small, fast, and they accurately mimic the characteristics of the original applications they are generated from. They can be used for early performance studies of multicore systems in both actual hardware and simulation. We implemented our approach in fully automated frameworks for CPUs and GPUs. Our fully automated synthetic benchmark generation frameworks comprise of two main steps: (1) characterizing an application to capture its inherent characteristics and modeling the captured application characteristics by an abstract benchmark model, (2) generating a synthetic benchmark using the abstract benchmark model.

Application characteristics can be divided into microarchitecture independent characteristics such as instruction mix, instruction level parallelism, data locality, thread communication; or microarchitecture dependent characteristics such as branch miss prediction and cache miss rate. In fact, significant work has been done to characterize single threaded CPU benchmarks [9, 10]. Although, there has been work in multi-threaded program characteristics such as memory level parallelism, ultimately the synthetic CPU benchmark generated by using these characteristics is a low level program. However, the development of synthetic CPU benchmarks demands high level characteristics since our goal is to develop synthetic CPU benchmarks suitable for any given infrastructure and low level characteristics simply do not allow the portability that we require.

As a solution, we detect these high level parallel pattern characteristics in developing synthetic multicore benchmarks. Patterns, in our case, parallel patterns, help ease the burden of parallel programming by bringing best practices to commonly occurring programming challenges. Parallel patterns are high level characteristics that define the structure of a multicore application in terms of data sharing, thread communication, general threading behaviors. They provide a way to design and create robust and understandable parallel multicore applications rapidly. Hence, using these high level parallel patterns in multicore benchmarks is crucial. Furthermore, detecting parallel patterns used in applications provides performance improvements and enables many architectural optimizations; however, this topic has not been widely studied. Nevertheless, manual detection of parallel patterns is an expensive and time-consuming task, which is becoming prohibitive with the increasing size of multi-threaded applications [11, 12]. Hence, we use machine learning techniques, which are often feasible and cost-effective for classification [13, 14], to automatically detect parallel patterns in these applications. Specially, we implemented k-nearest neighbor, decision trees, naive Bayes classifier, neural networks, and principal component analysis techniques for this purpose. The characteristics that we use to capture the behavior of real GPU applications are instruction throughput, compute unit occupancy, computation-to-memory access ratio, memory instruction mix, and memory efficiency. These characteristics are widely used in the literature [15–19].

Once, we capture the characteristics of an original application, we generate a synthetic benchmark from these characteristics. Note that we also use parallel pattern characteristics captured from the original application in developing synthetic CPU benchmarks. Our synthetic CPU benchmarks are in a high level programming language, C, as opposed to assembly in earlier works. Also, they can be generated using either Pthreads or Multicore Associations (message passing and resource management) libraries [20]. On the other hand, the synthetic GPU benchmark consists of host (CPU) and compute device (GPU) code and it is generated in C++ using OpenCL library. Our synthetic benchmarks preserve all of the application-level characteristics captured from the original application. Furthermore, to obtain high-fidelity synthetic benchmarks, we generate thread-level synthetic CPU benchmarks where they preserve the characteristics of individual threads by using hardware performance counter results for each thread. We show that thread-level synthesis is more challenging than application-level synthesis since application-level synthetics preserve aggregate characteristics whereas thread-level synthetics preserve per thread characteristics and the behavior of each thread impacts each other's behavior.

1.1. Contributions

This research presents an automatic synthetic benchmark generation framework for multicore systems including CPUs and GPUs. This thesis demonstrates that our synthetic benchmarks are smaller and faster than the original applications that they are derived from. That is, we show that generating synthetic benchmarks for target applications is a good solution to speed up architectural simulation of modern CPU and GPU architectures. Also, they are portable and human readable and they do not compromise the proprietary nature of the original applications. That is, the synthetic benchmarks have no functionality and cannot be reverse engineered to obtain the original applications. Furthermore, we leverage software architectural patterns in developing synthetic benchmarks for multicore CPU systems. To the best of our knowledge, this is the first time software architectural patterns are used to benchmark synthesis. This thesis makes the following major contributions.

Synthetic Benchmark Generation for Multicore CPUs (first published in [21–23]):

- The key novelty of our approach is that we use parallel patterns in generating synthetic multicore applications.
- We formalize parallel pattern recognition process by presenting reference behaviors for each parallel pattern type.
- We present an algorithm for synthetic multicore benchmark generation and developed MINIME tool.
- Our synthetics are portable since they are generated in a high level programming language, C, as opposed to assembly in earlier works. Also, they can be generated using either Pthreads or MCA libraries.
- We show that synthetic benchmarks are representative across a range of multicore machines with different architectures, while being on average 21× faster and 14× smaller than original benchmarks.

Thread-Level Synthetic Benchmarks for Multicore CPUs (first published in [24]):

- We present an algorithm for thread-level synthetic multicore benchmark generation.
- We compare application-level and thread-level synthetic benchmark generation results where thread-level synthetic benchmarks are more similar to the original benchmark that they are generated from.
- We demonstrate that we can generate multi-threaded synthetic benchmarks for real-life PARSEC and Rodinia benchmarks, while being faster (on average 147×) and smaller (on average 11×) than originals.

Synthetic Benchmark Generation for GPUs (first published in [25]):

- We use principal component data analysis methodology to identify critical GPU application characteristics.

Table 1.1. Our synthetic benchmarks.

	Synthetic Benchmark		
	Application-level CPU	**Thread-level CPU**	**GPU**
Input	PARSEC, Rodinia (OpenMP), and EEMBC MultiBench suites in C using Pthreads	PARSEC and Rodinia (OpenMP) suites in C using Pthreads	AMD APP SDK benchmarks in C/C++ using OpenCL
Output	Synthetic benchmarks in C using Pthreads, MCAPI or MRAPI	Synthetic benchmarks in C using Pthreads, MCAPI or MRAPI	Synthetic benchmarks in C/C++ using OpenCL
Avg. app-level similarity	92%	93%	96%
Avg. thread-level similarity	44%	84%	-
Avg. speedup	21×	147×	541×
Avg. code size reduction	14×	11×	1×

- A synthetic benchmark generation framework is proposed to generate synthetic OpenCL benchmarks for GPUs from a given GPU application.
- We implemented our solution in MINIME-GPU tool.
- The experimental results showed that our synthetic benchmarks mimic the characteristics of the original applications they are generated from across different architectures where the average similarity is 96% and average speedup is 541×.

Using Machine Learning Techniques to Detect Parallel Patterns of Multi-Threaded Applications (first published in [26]):

- We apply machine learning techniques in a novel approach to automatically detect parallel patterns and we compare these techniques in terms of accuracy and speed.
- We demonstrate that k-nearest neighbor, naive Bayes classifier, and decision trees are the most accurate techniques with a 100%, 96%, and 92% accuracy, respectively.
- We show that decision trees is the fastest technique where they provide a 5.7× average characterization speedup over the other techniques that do not use feature selection and a 4.8× speedup over the other techniques that use feature selection.

Overall, Table 1.1 summarizes the synthetic benchmarks we generate in this thesis. In the table, we show the inputs, outputs, and experimental results including average application-level similarity, thread-level similarity, speedup obtained in terms of execution time, and code size reduction in lines of code going from the original to the synthetic for our application-level CPU, thread-level CPU, and GPU synthetic benchmarks. Note that thread-level similarity is not applicable for synthetic GPU benchmarks and also there is no reduction in code size for synthetic GPU benchmarks since original GPU applications are already small.

1.2. Organization

This thesis is organized as follows. We present some background information on software architectural patterns, programming models for CPUs and GPUs, machine learning models, and benchmarks for performance evaluation in Chapter 2. We show how we automatically generate synthetic benchmarks for multicore CPUs in Chapter 3. Chapter 4 describes our technique for thread-level synthetic benchmark generation for multicore CPUs. In Chapter 5, we present our synthetic benchmark generation technique for GPUs. We give the details of the machine learning techniques we used to detect parallel pattern of multi-threaded applications in Chapter 6. We discuss related work in Chapter 7. We provide conclusions and directions for future work in Chapter 8.

2. BACKGROUND

In this chapter, we present some background information on software architectural patterns, programming models for CPUs and GPUs, machine learning techniques, and benchmark suites. We rely on this background information to explain our benchmark synthesis and pattern detection techniques in the thesis.

2.1. Software Architectural Patterns

Architectural patterns are fundamental organizational descriptions of common top-level structures observed in a group of software systems [27]. One of the most important decisions during the design of the overall structure of a software system is the selection of an architectural pattern. Architectural patterns allow software developers to understand complex software systems in larger conceptual blocks and their relations, thus reducing the adoption complexity and providing less error prone applications.

Architectural design patterns have been developed for object-oriented software and have been found to be very useful [28]. Similarly, a parallel pattern language which is a collection of design patterns, guiding the users through the decision process in building a parallel system has been developed [29]. In a pattern language, patterns are organized into a hierarchical structure so that the user can design complex systems going through the collection of patterns. A parallel pattern language also provides domain-specific solutions to the application designers in less time. In this thesis, we use the term parallel pattern as synonym for parallel software architectural pattern.

There exist three classes of parallel patterns based on organization of tasks, data, and flow of data. Figure 2.1 shows parallel patterns in a decision tree [30]. We detect these parallel patterns as described in Chapter 6 and then we use them in benchmark synthesis as described in Chapters 3 and 4. Each parallel pattern has unique architectural characteristics to exploit. When a work is divided among several independent tasks, which cannot be parallelized individually, the parallel pattern employed is *Task*

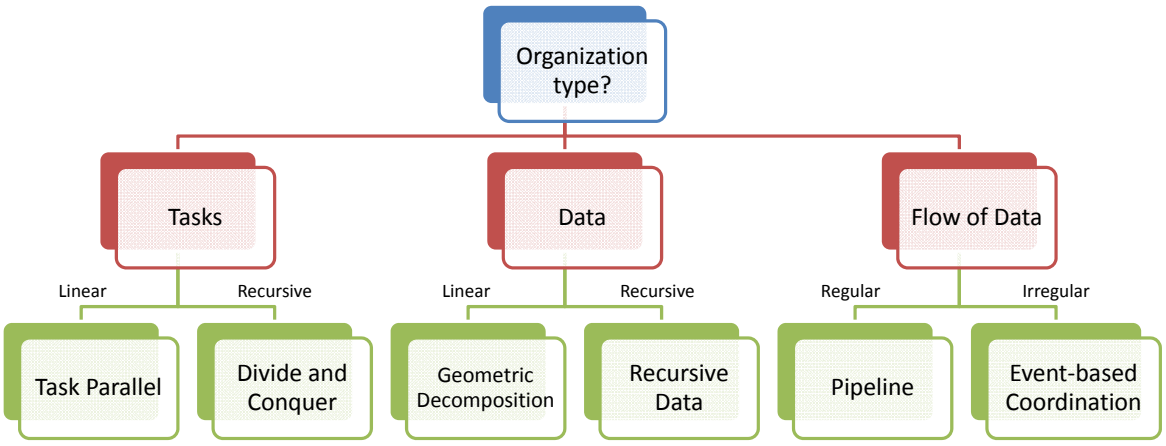

Figure 2.1. Parallel patterns for software.

Parallelism (TP). The independent tasks may read shared data, but they produce independent results. In *Divide and Conquer (DaC)*, a problem is structured to be solved in sub-problems independently, and merging the outputs later. This pattern is used to solve many sorting, computational geometry, graph theory, and numerical problems. Divide and conquer algorithms can cause load-balancing problems when using non-uniform sub-problems, but this can be resolved if the sub-problems can be further reduced.

In data centric patterns, data is decomposed aligned with the set of tasks. When the data decomposition is linear, the parallel pattern that is employed is called *Geometric Decomposition (GD)*. In GD, data decomposition can inherently deliver a natural load balancing process since data is partitioned into equal size. Matrix, list, and vector operations are examples of geometric decomposition. Parallel pattern used with recursively defined data structures is called *Recursive Data (RD)*. Graph search and tree algorithms are example usages of recursive data.

Apart from task parallelism and data parallelism, if a series of ordered but independent computation stages need to be applied on data, where each output of a computation becomes input of a subsequent computation, *Pipeline (Pl)* parallel pattern is used. Each stage processes its data serially and all stages run in parallel to increase the throughput. *Event-based Coordination (EbC)* parallel pattern defines a set of tasks that run concurrently where each event triggers starting of a new task. In

this pattern, the interaction can take place at irregular and unpredictable intervals.

In a multi-threaded application that uses parallel patterns, generally a big problem is divided into sub-problems. In these applications, execution starts on the main thread and the main thread creates worker threads for solving sub-problems in parallel. In TP, communication is low and each thread can work on problems with different sizes. The problem is divided into sub-problems in DaC, so the worker threads can solve the sub-problems independently with few communications. In GD, each worker thread works on one part of a big data with many communications. RD is similar to GD but the big data such as a graph is not partitioned equally. In Pl, each worker thread does some work and passes the partial result to the worker thread in the next stage. This results in few communications between threads. When the interactions between stages are not feed-forward, we have EbC.

The above architectural patterns capture the essence of multicore applications at a high level. This concept has not been used in synthetic benchmark generation before us and allows us to have portable benchmarks that preserve both high level and low level characteristics.

2.2. Multicore Programming APIs

Multicore programming Application Programming Interfaces (APIs) reduce the complexity involved in writing software for multicore systems. Since multicore programming APIs standardize task management, communication, and resource sharing, they establish portability and makes it easier to reuse the application across different multicore platforms. Utilizing multicore programming APIs, system developers can write portable programs that can scale throughout current and future generations of multicore processors and architectures, benefiting application, processor and system developers. Multicore programming APIs also allow chip vendors and third-party tool providers to take over the resource management, so programmers can focus on high-level applications. While generating synthetic benchmarks as described in Chapters 3 and 4, we support multiple multicore programming APIs such as POSIX, Multicore

Resource Management API (MRAPI), and Multicore Communication API (MCAPI). Note that while POSIX targets general purpose computing, MCAPI and MRAPI target embedded computing for closely distributed systems. Since we generate synthetic benchmarks for embedded systems, we use MCAPI and MRAPI apart from POSIX. We now describe these multicore programming APIs in more detail.

POSIX API is a well-known API that defines a standard operating system interface and environment to support applications portability at the source code level. We use Pthreads library from POSIX API because Pthreads is an accepted and widely used standard for shared memory in multicore systems. The Pthreads library is a set of C programming language types and procedure calls. This library provides functions for creating/destroying threads, and for coordinating threads while accessing shared resources. Pthreads library supplies locks (mutexes, semaphores), conditions variables to use while coordinating threads. Threads can also read/write global data as well as shared memory.

MCAPI [20] is a message-passing API that aims to supply communication and synchronization between closely distributed multicore embedded systems. Whereas the Message Passing Interface (MPI) [31] supplies communication in widely distributed systems such as computer networks and does not reflect anything about cores at the programming level. Since we target multicore systems we use MCAPI that provides scalable, high performance, low latency, and low overhead communication for heterogeneous platforms (in terms of core, interconnect, memory, operating system, software tool-chain, and programming language). MCAPI has three fundamental communication types: connectionless datagrams for messages; connection-oriented, unidirectional, FIFO packet streams for packet channels; and connection-oriented single-word unidirectional, FIFO packet streams for scalar channels. Basic elements of the MCAPI topology are nodes, which can be a process, a thread, or a hardware accelerator. Communication occurs between endpoints, which are termination points and created on nodes on each side of the communication. Both connectionless and connection-oriented communications take place between endpoints. MCAPI provides sufficient number of functionalities while hiding or minimizing communication overhead to get better performance.

MCAPI also provides source-code compatibility that allows multicore applications to be ported from one operating environment to another.

MRAPI [20] is a resource management API that manages synchronization constructs such as mutexes, semaphores reader/writer locks, shared and distributed memory regions, and hardware information available at run-time called metadata in closely distributed multicore embedded systems. Note that Pthreads library also provides similar functionalities but Pthreads library relies on the presence of a multicore operating system. Since we target multicore embedded systems where an operating system or Pthreads library support cannot be present, we use MRAPI in our synthetic benchmarks. MRAPI provides the ability to declare and allocate/destroy shared memory regions and to identify nodes which have access to each region. MRAPI provides application-level synchronization primitives for coordinating concurrent access to shared resources (homogeneous or heterogeneous cores or chips, hardware accelerators, memory regions). MRAPI also supports the ability to create/destroy and manage synchronization constructs. Similar to MCAPI, MRAPI is scalable and provides source-code compatibility that allows multicore applications to be ported from one operating environment to another.

2.3. GPUs and Programming Models

In this section, we provide a summary of the background information required to understand benchmark characterization and synthesis for GPUs described in Chapter 5.

2.3.1. OpenCL Programming Model

Open Computing Language (OpenCL) is an open standard based upon C for portable parallel applications across heterogeneous platforms including CPUs, GPUs, and DSPs [32]. OpenCL provides an API to develop parallel applications using task-based and data-based parallelism. Figure 2.2 shows OpenCL platform model [32] where an OpenCL platform has a *host* connected to a number of *compute devices*. Note that

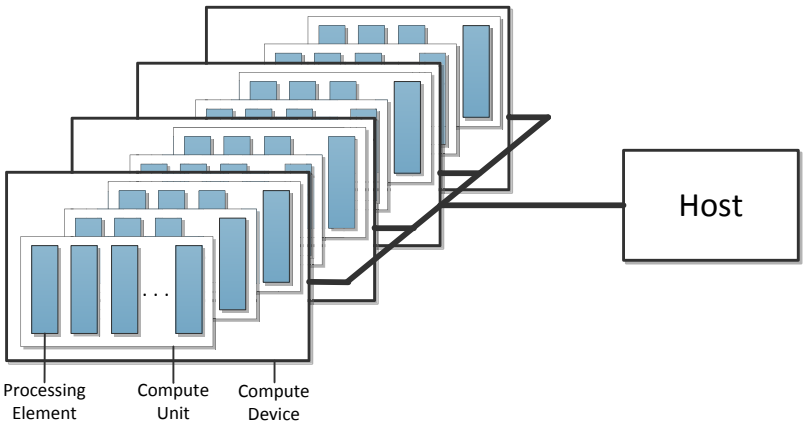

Figure 2.2. OpenCL platform model.

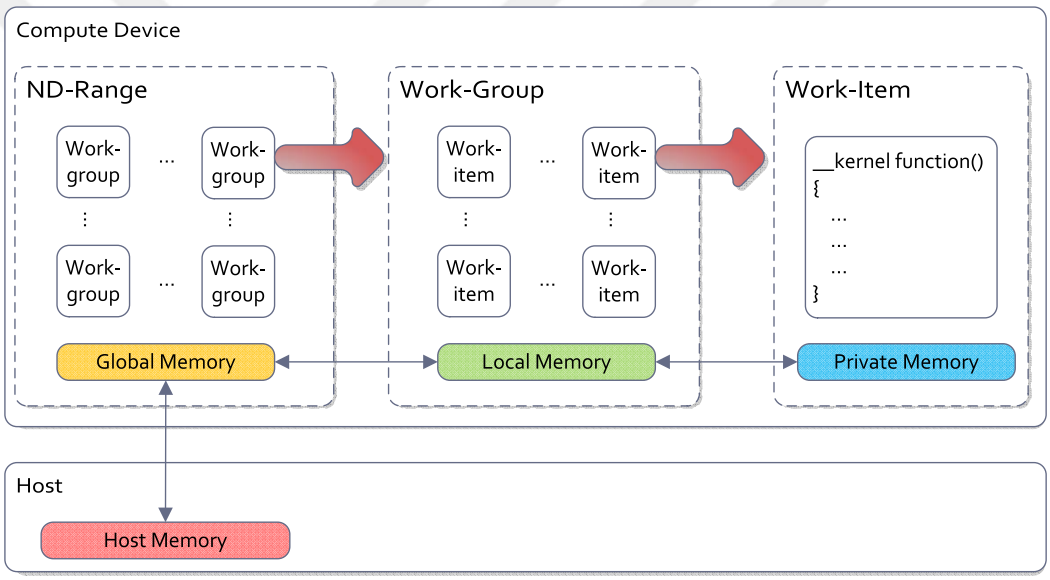

Figure 2.3. OpenCL programming and memory model.

the host is the CPU that submits works to the compute devices. A compute device consists of one or more *compute units* (cores) where a compute unit is composed of a set of *processing elements*.

Figure 2.3 shows OpenCL programming and memory model [1]. OpenCL (programming model) allows developing OpenCL programs, which requires developing codes for the host side (host program) and device side (kernel program) as well. The kernel program has parallel functions called *kernels*, which are the basic units of executable code. The host is developed in C/C++ using OpenCL API (library) and it manages the device to execute the kernel program. When a host program invokes a

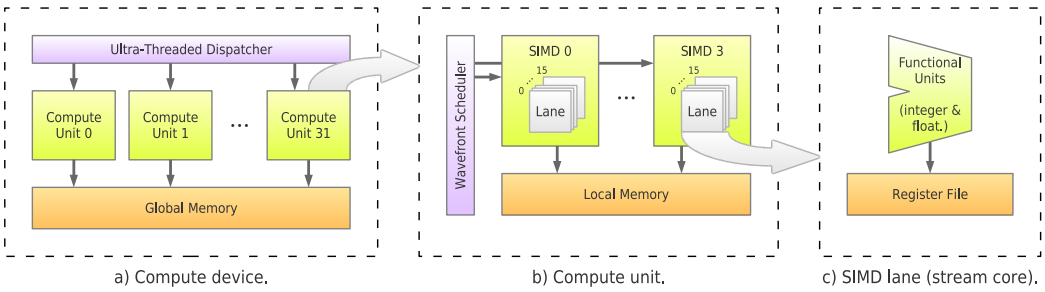

Figure 2.4. Simplified AMD Southern Islands GPU architecture [1].

kernel, an index space called *N-Dimensional Range (NDRange)* that can be arranged into 1, 2 or 3 dimensions is defined. An NDRange consists of *work-items* and a number of work-items are organized into a *work-group*. Note that each work-item executes the same kernel (usually on different data). OpenCL provides the notion of dimension to define the number of work-items where *global dimensions* define the range of computation (whole computation space) and *local dimensions* define the size of the work-groups. OpenCL memory model defines a three-level memory hierarchy for a compute device. All work-items can access a *global memory*, work-items in a single work-group share a *local memory*, and every work-item has a *private memory* (registers) that is not accessible from other work-items. Note that synchronization is allowed between work-items within a work-group and there is no synchronization between work-groups.

2.3.2. Target GPU Architecture

In this thesis, we target the most recent Advanced Micro Devices (AMD) Southern Islands (SI) GPUs and Figure 2.4 shows a simplified diagram of this architecture [1]. In the OpenCL terminology, we denote AMD SI GPU as the compute device, SIMD units as compute units, and SIMD lanes as processing elements. AMD SI has scalar and vector arithmetic-logic units and also it has multiple levels of memory. An ultra-threaded dispatcher in AMD SI schedules work-groups that are pending on the running ND-Range. Also, wavefronts that consist of 64 work-items are created to efficiently run the same code in a SIMD fashion.

2.4. Machine Learning Techniques

In this section, we provide information about machine learning techniques that we use to detect (classify) parallel patterns of multi-threaded CPU applications as shown in Chapter 6. Since we have predefined classes of parallel patterns as described in Section 2.1, we use classification techniques from machine learning algorithms instead of clustering techniques [13, 14]. Note that classification is the task to learn to assign data to predefined classes and in clustering, no predefined classification is required and large data sets are grouped according to their similarity. In classification, a model is constructed from a training set and then this model is used to classify unseen new data. Next, we briefly describe these classification techniques.

2.4.1. k-Nearest Neighbor

In k-Nearest Neighbor (kNN) [13, 14], the class of a data sample is assigned to the class that is most common among its k nearest neighbors. If k is 1, then the data sample is simply assigned to the class of its nearest neighbor. In general, Euclidean Distance measure is used to calculate how close the data samples are.

2.4.2. Decision Trees

A decision tree [13, 14] is a hierarchical tree structure that is constructed from known data samples, where an internal node of the tree denotes a test on a feature (sub-characteristic) and a leaf node represents a class. Since there can be continuous valued features in a data sample, these features are discretized during decision tree generation by splitting their range into two intervals. Once the decision tree is constructed, it is possible to use the decision tree to predict the classes of previously unseen data samples. Decision rules that provide unique paths for the sample data to the class that it belongs to are used to classify unseen data samples. A decision tree may overfit the training set and this can result in poor accuracy for unseen data samples. In this case, the tree is pruned by deleting nodes.

2.4.3. Naive Bayes Classifier

A naive Bayes classifier (NBC) [13, 14] is a statistical classifier that is based on Bayes' theorem and performs prediction of class membership probabilities. An NBC works by using a training set that is a set of data samples already associated to a class. Using the training set, the classifier computes for each feature (sub-characteristic) of a data sample, the probability that a data sample belongs to each of the considered classes. To calculate the probability that a data sample belongs to a class, the classifier multiplies together the individual probability of each of its features in this class. The class with the highest probability is the one the data sample is most likely to belong to. NBC assumes that all of the features of a data sample independently contribute to the probability that a data sample belongs to a class even if the features of a data sample depend on each other. When there exist continuous valued features in the data set, NBC uses probability distribution methods such as Gaussian to model continuous values.

2.4.4. Neural Networks

Neural networks (NNW) [13, 14] are computational models that are useful for pattern recognition and data classification. The objective of neural networks is to transform the given inputs into the desired outputs. A neural network includes a set of input/output units called neurons that are connected to each other and each connection has a weight. In a neuron, each input is multiplied by a connection weight and the products are summed. A transfer function takes the sum and generates a result for the next layer or a final output.

Neural networks are composed of an arbitrary number of neurons with an arbitrary topology. Neural networks have an input layer, an output layer, and one or more hidden layers between the input and output layers. While information travels only one direction in feedforward (topology) networks, information travels in both directions in feedback (topology) networks.

During the learning phase of a neural network, the performance of the neural network, which is the difference between the predicted and the actual output, is measured at each iteration. According to the performance of the neural network, the connection weights are updated using a training function. Once the neural network meets the target performance, it can predict the class label of unseen input data.

2.4.5. Principal Component Analysis with K-means

The performance of machine learning techniques can be poor, when dimensionality of data is high. There can be redundant dimensions (features) that represent very little information. Hence, reducing the number of the features can improve the performance of machine learning techniques. Principal Component Analysis (PCA) [33] is a statistical technique to find the major contributors to the variance of the data. PCA reduces dimension of the data and projects the high dimensional data into a low dimensional, in general, 2 or 3 dimensional, sub-space. This projection preserves the highest variance observed in the data on the first dimension, the second highest variance on the second dimension, and so on.

The idea is that a combination of PCA and k-means clustering [34] can be used for data classification since the new low dimensional sub-space preserves much of the original variance and expresses the original data. In this technique, k-means clustering is performed on the sub-space, after generating the PCA sub-space. The k-means algorithm clusters data samples based on their principal components into k partitions and finds the centroid of each cluster of data. The algorithm calculates the distance, such as Euclidean distance, between a data sample and the cluster centroids and assigns the data sample to the closest cluster. Cluster centroids are then re-calculated according to new class memberships. This process continues until no more changes in class memberships are done. The output of the k-means algorithm is the cluster centroids and the cluster memberships of data samples. Once the cluster centroids are found, the class of a new unseen data can be determined by using a classification algorithm.

2.5. Benchmarks for Performance Evaluation

Benchmarking is one of the most important methods that is used for performance evaluation of computer system designs [35]. In benchmarking, the execution of the benchmark applications, which represent the real-life applications of interest, are studied. It is crucial that the behaviors (characteristics) of the selected benchmarks accurately cover the behaviors of the applications of interest as well as the target system. For example, one needs multi-threaded benchmarks to analyze multicore systems.

A benchmark suite includes a set of benchmarks to represent a wide range of real-life applications and targets specific systems. We perform experiments on several benchmark suites in order to validate our techniques. Since our goal is to generate synthetic benchmarks for multicore systems, we use parallel benchmark suites including PARSEC [3] that targets multi-threaded applications for shared memory architectures and Rodinia [4] that targets heterogeneous systems including GPUs. Also, note that there exist several benchmark suites of sequential applications. For example, SPEC CPU2006 (serial) [36] contains serial programs and does not target parallel machines and MiBench [37] targets uniprocessor embedded systems. In this section, we describe the characteristics of these parallel benchmark suites.

PARSEC is a well-known, open-source multi-threaded benchmark suite with fundamental parallelism constructs. It includes a diverse set of workloads from different domains such as media processing or financial analysis that mimic large-scale commercial workloads.

Rodinia is a benchmark suite for heterogeneous computing. To help developers study emerging platforms including CPUs and GPUs, Rodinia covers a wide range of parallel communication patterns, synchronization techniques, and power consumption behaviors. The benchmarks in the suite have been parallelized with OpenMP for multicore CPUs and with the Compute Unified Device Architecture (CUDA) and OpenCL APIs for GPUs.

EEMBC MultiBench [6] uses a thread-based API to establish a common programming model and targets the evaluation of scalable SMP architectures with shared memory. Individual benchmarks in EEMBC MultiBench target three forms of concurrency: data decomposition, multiple data stream processing, and processing of multiple workloads.

2.5.1. Synthetic Benchmarks

Synthetic benchmarks are simplified artificial applications that can represent real-life applications or benchmarks. One approach to generate synthetic benchmarks is deriving them from existing applications and another approach is generating them by varying application characteristics. In this approach, a synthetic benchmark is constructed to represent existing application characteristics (metrics) such as instruction mix, cache miss rate, and thread communication. Once the synthetic benchmark is generated from an original existing application, it's accuracy (similarity) is measured in order to validate the representativeness of the synthetic benchmark. We use the following definitions to measure the similarity between a synthetic benchmark and the original application that it is derived from.

Definition 2.1 (error rate)**.** *Given a similarity metric (characteristic), mt, and the value of mt for the synthetic and original application as mtsyn and mtorg, respectively, the error rate for mt is,* $errorrate_{mt} = |(mtsyn - mtorg)|/mtorg$.

Definition 2.2 (individual similarity score)**.** *The individual similarity score (*iss_{mt}*) is* $iss_{mt} = [1 - errorrate_{mt}] \times 100$ *and ranges from 0 to 100.*

Definition 2.3 (overall similarity score)**.** *The overall similarity score (oss) is* $oss = (iss_{mt-1} + iss_{mt-2} + \ldots + iss_{mt-N}) \,/\, N$*, where* N *is the number of the individual similarity scores.*

3. SYNTHETIC BENCHMARK GENERATION FOR MULTICORE CPUs

3.1. Overview

Multicore chips can allow higher performance at lower energy. However, development of new multicore systems requires a large number of benchmarks. At the same time, there is an increase in simulation runtimes of benchmarks that limits our ability to fully explore the design space. We need to develop faster benchmarks.

In order to develop a synthetic benchmark, the first step is to characterize the given application. Characterization consists of a description of the application by means of quantitative parameters and functions; the objective is to derive a model able to show, capture, and reproduce the behavior of the application and its most important features. Application characteristics can be divided into microarchitecture independent characteristics such as instruction mix, instruction level parallelism, data locality, thread communication; or microarchitecture dependent characteristics such as branch miss prediction and cache miss rate. In fact, significant work has been done to characterize single threaded benchmarks [9, 10]. Although, there has been work in multi-threaded program characteristics such as memory level parallelism, ultimately the synthetic benchmark that has so far been developed in the literature is a low level program unlike ours. The development of synthetic benchmarks for multicore systems demands high level characteristics since our goal is to develop synthetic benchmarks suitable for any given infrastructure and low level characteristics simply do not allow the portability that we require.

In order to solve above limitations, we need to develop new benchmarks but benchmark development process is time- and labor-intensive. We present a novel synthetic benchmark synthesis approach using parallel patterns that addresses these limitations. Synthetic benchmarks do not perform any useful computation, yet they can

approximate characteristics of real-life applications. These benchmarks can be generated by varying application characteristics or can be derived from existing benchmarks. A synthetic benchmark is smaller and faster than the original benchmark that it is derived from hence it simulates faster. In this chapter, we generate synthetic multicore benchmarks that are fast, portable, and suitable for any given infrastructure.

We experimentally validate our techniques by generating synthetic multicore benchmarks from PARSEC, Rodinia, and EEMBC benchmark suites using our MINIME tool. Our synthetics can use either Pthreads or Multicore Association (MCA) libraries [38], the latter allowing us to have infrastructure independent benchmarks. Synthetic benchmarks are compared with the original benchmarks using similarity metrics based on both microarchitecture dependent and independent characteristics. We found that synthetic benchmarks using Pthreads library are similar on average 92% to the original benchmarks and our benchmarks using MCA libraries are similar up to 95% and on average above 86%. We also found that the synthetics that correctly captured the parallel patterns in the originals have a high level of similarity.

We first published our results on synthetic CPU benchmark generation in [21–23]. In particular, this chapter makes the following contributions.

- The key novelty of our approach is that we use parallel patterns in generating synthetic multicore applications.
- We formalize parallel pattern recognition process by presenting reference behaviors for each parallel pattern type.
- We present an algorithm for synthetic multicore benchmark generation for CPUs.
- Our synthetics are portable since they are generated in a high level programming language, C, as opposed to assembly in earlier works. Also, they can be generated using either Pthreads or MCA libraries.
- Our synthetics are suitable for embedded systems and can run on any given infrastructure thanks to using MCA libraries.
- Our synthetics can act as proxies for proprietary customer applications that are not publicly available.

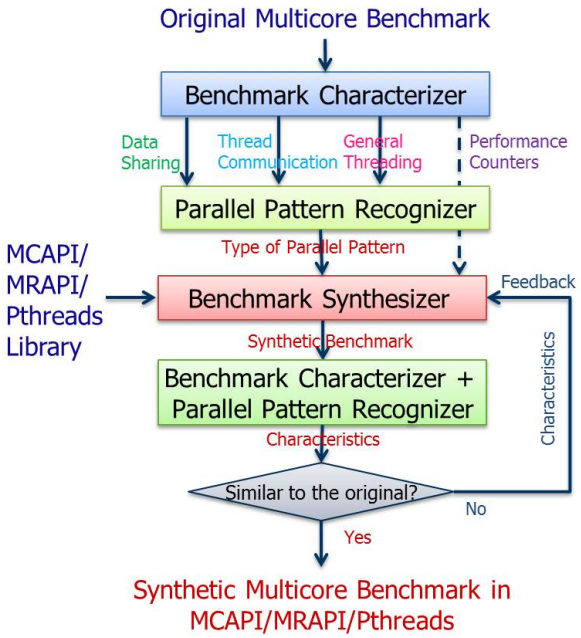

Figure 3.1. MINIME: Pattern-aware multicore benchmark synthesizer architecture.

- We developed MINIME tool and experimentally validate our techniques on both x86 and Power Architecture systems using PARSEC, Rodinia and EEMBC Multi-Bench benchmark suites. Experiments show that our synthetics are similar with the originals with respect to several metrics. They are also faster and smaller than originals and they mimic the behavior of the original on different microarchitectures.
- We study the impact of input changes on the synthetics. We also perform correlation studies to determine the importance of correct parallel patterns in achieving high similarity.
- We show that synthetic benchmarks are representative across a range of multicore machines with different architectures, while being on average 21× faster and 14× smaller than original benchmarks.

3.2. High-level Framework

Figure 3.1 shows a high level view of our fully automated framework MINIME. Our tool contains three main modules: *benchmark characterizer*, *parallel pattern recognizer*, and *benchmark synthesizer*. In benchmark characterizer module, we derive a

model that can capture, and reproduce the behavior of the original application and its most important features. Then, in pattern recognizer module, we detect (recognize) the parallel pattern of the original application from the captured characteristics. Lastly, in benchmark synthesis module, we generate a synthetic benchmark by using the derived model and detected parallel pattern. Next, we explain the benchmark characterizer, parallel pattern recognizer, and benchmark synthesizer modules in detail.

3.3. Multicore Benchmark Characterization

We use both microarchitecture independent and dependent characteristics to obtain characteristics of an application. These characteristics form an abstract benchmark model. Our abstract benchmark model captures the important high level and low level application characteristics that potentially impact an application's performance and architectural pattern. Note that at the cost of slightly reduced accuracy, our benchmark model captures an application behavior with just a few microarchitecture independent characteristics as compared to previous works [39, 40] that use a high number of such characteristics. As can be seen in Table 3.1, we analyze multicore benchmark characteristics in four groups: *Data Sharing (DS), Thread Communication (TC), General Threading (GT)*, and *Performance*, where each group also has sub-characteristics. Our data sharing, thread communication, and general threading characteristics are similar to [2]. We formalize them and add a new group of performance characteristics here. While data sharing, thread communication, and general threading groups include high level (software architectural) sub-characteristics, performance group includes low level (non-software architectural) sub-characteristics except communication to computation ratio. We now describe each group in more detail.

3.3.1. Data Sharing Characteristics

Sub-characteristics in the data sharing group are *private, read-only, producer/-consumer*, and *migratory*, similar to [2, 41]. We have formalized this concept in this thesis as follows. We use $\#readers$ and $\#writers$ to indicate the unique number of threads that read or write the same cacheline, respectively. We use cachelines as our

Table 3.1. CPU benchmark characteristics.

Characteristics	**Sub-characteristics**	**Level**
Data Sharing	Private Read-only Producer/Consumer Migratory	High
Thread Communication	None Few Many	High
General Threading (per thread)	Program Counter (PC) Dynamic instruction count Creator thread Creation time Exit time Lifetime	High
Performance	Instructions Per Cycle (IPC) Cache Miss Rate (CMR) Branch Misprediction Rate (BMR)	Low
	Communication to Computation Ratio (CCR)	High

technique uses binary instrumentation that models memory accesses per cacheline.

Definition 3.1 (private)**.** *If* $\#readers = \#writers = 1$*, the data has* private *sub-characteristic.*

Definition 3.2 (read-only)**.** *If* $\#readers > 0$ *and* $\#writers = 0$*, the data has* read-only *sub-characteristic.*

If multiple threads access the same data and at least one operation is write then this means that the data is shared between threads. Shared can be classified into two sub-characteristics.

Definition 3.3 (producer/consumer)**.** *If* $\#readers > \#writers$*, the data has* producer/consumer *sub-characteristic.*

Definition 3.4 (migratory)**.** *If $\#readers \leq \#writers$, the data has* migratory *sub-characteristics, where a thread reads and writes to a shared data item and this behavior is repeated by many threads.*

3.3.2. Thread Communication Characteristics

Sub-characteristics in the thread communication group are *none*, *few*, and *many*, similar to [2]. We have formalized this concept in this thesis as follows. The ratio of cachelines used for communication to all cachelines used during execution gives shared cachelines, denoted by $sharedCL$. We use $numTH$ to denote the total number of threads during execution and $commTH$ to denote pairwise communicating threads and can be at most $numTH^2$.

Definition 3.5 (none)**.** *When ($commTH < numTH$) and ($sharedCL \leq 0.3$), thread communication characteristic of an application is* none.

Definition 3.6 (few)**.** *When ($commTH = numTH$) and ($0.3 \leq sharedCL \leq 0.8$), the thread communication characteristic is* few.

Definition 3.7 (many)**.** *When ($commTH > numTH$) and ($0.8 \leq sharedCL \leq 1$), the thread communication characteristic is* many.

We experimentally determine 0.3 and 0.8 where these values give the most accurate thread communication characteristics. For example, while threads can communicate in any direction in geometric decomposition, there exists a communication from a thread in stage i to a thread in stage $i + 1$ in pipeline pattern.

3.3.3. General Threading Characteristics

We keep a track of the following general threading sub-characteristics for each thread. These are *Program Counter (PC)*, *dynamic instruction count (IC)*, *creator thread*, and *creation/exit time*. First, we use the program counter to indicate the starting program counter of a thread. Threads with the same program counter execute the

same function, whereas threads with different program counters each execute a unique function. Second, we use dynamic instruction counts of threads to decide whether threads are balanced or not, since load balancing is important for better utilization of multicore hardware. Third, we build a creator-child graph of threads by using the creator thread information. To label threads in this graph, we use Pthreads [42] terms, namely, creator thread, main thread, and worker thread. In the case of OpenMP [43] applications, we assume that the compiled code uses Pthreads library. We label a thread that calls the `pthread_create()` function as the creator thread of the new child thread. The main thread is the thread that is the first thread in the process and runs the `main()` entry function. The worker thread is used to describe all threads except for the main thread. A main thread or a worker thread can also be a creator thread. For example, a master thread in OpenMP, which can be a main thread or a worker thread, can create a number of worker threads that execute blocks of code in parallel. Fourth, we collect creation and exit times of threads and calculate the *lifetime (LT)* sub-characteristics of threads. Similar lifetimes can point to balanced threads, and creation times are used to decide whether threads are created dynamically. Since applications with the same number of instructions can have different lifetimes due to different instruction mixes, the lifetime sub-characteristics cannot be approximated with the dynamic instruction count.

3.3.4. Performance Characteristics

Sub-characteristics in the performance group are microarchitecture dependent characteristics including *Instructions Per Cycle (IPC), last level Cache Miss Rate (CMR), and Branch Misprediction Rate (BMR)* and microarchitecture independent characteristics including *Communication to Computation Ratio (CCR)*. IPC is the ratio of the number of instructions retired to clock cycles. CMR is the ratio of the last level cache demand requests that missed the last level cache to the last level cache demand requests. BMR is the ratio of the number of mispredicted branches retired to the number of branch instructions retired. CCR is the ratio of communication instructions (load and store) to the computation instructions (instructions except load

and store). These metrics are used commonly in the literature to assess performance of applications.

3.3.5. Characterization Tools

We use a dynamic binary instrumentation tool, named DynamoRIO [44], which is similar to Pin [45], for gathering above-mentioned high level characteristics during the execution of an application. DynamoRIO is an open source run-time code manipulation system that supports code transformations on any part of an application, while it executes. We also use Umbra [46], which is an efficient and scalable memory shadowing tool built on top of DynamoRIO. We developed our characterizer as a client of DynamoRIO and Umbra. Note that our tool instruments applications at the binary level which allows us to work with legacy/proprietary Intellectual Properties (IPs) and not compile/link applications. Our client analyzes the data sharing pattern of the application by observing cacheline accesses. Similarly, we use our client to determine thread communication between threads. We dynamically build a thread communication matrix during the execution of an application, where two threads are communicating if one thread writes to a cacheline and the other one reads from the same cacheline. For tracking general threading information, our client wraps thread creation operations, detects the program counter, dynamic instruction count, creator, and creation/exit time of each thread. We also used perf tool (Linux profiling with performance counters) [47] to obtain microarchitecture dependent characteristics. The perf tool is a kernel-level subsystem that provides a framework for collecting performance data. It can be used to measure one or more hardware events including instructions, cycles, cache misses, and branch misses. For instance, perf tool can compute the IPC from a process's counts of instructions and cycles.

3.4. Parallel Pattern Recognition

Once we obtain the characteristics of a given application, we use machine learning techniques to automatically detect parallel patterns. In this chapter, we use k-Nearest Neighbor (kNN) classification where k is 1 to decide the parallel pattern of the appli-

cation. In addition, we use other machine learning techniques including decision trees and neural network to detect parallel patterns as we will describe in Chapter 6. In kNN, there exist two steps, which are the construction of a classification model and the usage of the model. In the first step, we use a set of reference behaviors that capture the key characteristics that each parallel pattern exhibits in order to construct a model that recognizes the parallel patterns described above. We use all high level characteristics except CCR given in Table 3.1 as the key characteristics. We do not use CCR because data sharing and thread communication characteristics implicitly cover this information.

We describe the reference behaviors for each group of characteristics and for each parallel pattern in Table 3.2 which is similar to the work in [2], Table 3.3, and Table 3.4. The table entries are either empty, or they contain single or multiple stars, where the higher number of stars denote the higher likelihood of the corresponding pattern to exhibit the sub-characteristics. For example, in Table 3.4 all sub-characteristics are most similar for threads in an application with GD pattern. We developed our reference behaviors for each group of characteristics by investigating the behavior of threads for each pattern type in the literature [30]. We then experimentally validated that these characteristics and reference behaviors are indeed observed in multi-threaded applications for which we knew the pattern for. Note that these applications were not used in the experiments.

In the second step, we measure the Euclidean distance between each group of characteristics of the application and characteristics of the reference behavior defined in the model. The scores of the parallel patterns are assigned to be inversely proportional to the distances to the application characteristics for that group, where the highest score is 100 (for the closest) and the lowest score is 0 (for the farthest). Hence, we calculate data sharing score, thread communication score, and general threading score for each type of pattern. We then sum the scores for each parallel pattern and the parallel pattern with the highest total pattern score gives the parallel pattern of the application.

Table 3.2. Data sharing reference behavior [2].

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

Table 3.3. Thread communication reference behavior.

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

Table 3.4. General threading reference behavior.

	TP	DaC	GD	RD	Pl	EbC
PC	***	**	****	****		
Dyn. Inst. Count	**	*	****	**	**	***
Creator	****	**	****	***	***	*
Creation Time	***	*	****	**	***	**
Exit Time	**		***		**	*
Lifetime	**	**	****	*	***	**

We now give examples of parallel pattern recognition. The parallel pattern of an application with read-only and private data sharing characteristics, and no inter-thread communication, where threads have unique PCs, and threads are created by the same thread at the beginning of the application is task parallel. As a real example we can use an image recognition application in which four separate identification tasks share the same input image data and each task is specialized to identify different objects such as people or place in the image. An example of geometric decomposition pattern can be an application with many producer/consumer and few migratory data sharing char-

Table 3.5. Algorithm to generate the code for a thread t_i based on parallel pattern.

Input	Thread t_i, Characteristics and parallel pattern of the original application, library of the synthetic							
Output	Code block for thread t_i							
Algorithm Steps	**Step**	**Operation**	**TP**	**DaC**	**GD**	**RD**	**Pl**	**EbC**
	Step 1	Create communication objects		X	X	X	X	X
	Step 2	Perform initial computation operations		X				X
	Step 3	Create child threads, if they exist in the original application		X		X		X
	Step 4	Get/Open communication objects		X		X		X
	Step 5	Begin loop, if pattern is pipeline					X	
	Step 6	Perform initial communication operations		X	X	X	X	X
	Step 7	Perform internal computation operations	X	X	X	X	X	X
	Step 8	Perform final communication operations		X	X	X	X	X
	Step 9	End loop, if pattern is pipeline					X	
	Step 10	Wait for child threads, if they exist in the original application		X		X		X
	Step 11	Perform final computation operations		X				X
	Step 12	Delete/Release/Close communication objects		X	X	X	X	X

acteristics, many data dependent inter-thread communication, and balanced threads that share the same PC and created by the main thread at the same time.

3.5. Pattern-Aware Synthetic Benchmark Generation

We iteratively generate the synthetic benchmark code in a fully automated manner without any user intervention. The iterations continue until thresholds for individual similarity scores and overall similarity score are satisfied or the user defined threshold for the number of iterations is reached. The iterative process includes code generation for high level metrics, similarity measurement between the original application and the synthetic benchmark, and code generation for low level metrics based on characteristics of the original application. Next, we describe each step in details.

3.5.1. Code Generation for High Level Metrics

The generated code consists of a main function and a function for each task where a task can be executed by one or more worker threads that are spawned from any thread using the *pthread_create()* function call. We apply the algorithm given in Table 3.5 to generate the function of each (worker) thread, t_i. The characteristics,

hence the parallel pattern of the original application, the particular thread t_i, and the library type (Pthreads, MCAPI, or MRAPI) are given as inputs to the algorithm. The output of the algorithm is the code block for thread t_i. Each step of the algorithm defines an operation and whether the operation in that step is performed for that parallel pattern type (denoted by X).

We also demonstrate our algorithm on a matrix multiplication application in Section 3.5.4 and generate the synthetic given in Figure 3.4. The synthetic is commented with the corresponding algorithm steps. Since the parallel pattern of the matrix multiplication application is geometric decomposition, we perform the steps given in column *GD* of Table 3.5.

- *Step* 1: Thread t_i creates communication objects that are used for data sharing. Communication objects include shared memory, semaphores, mutexes for Pthreads/MRAPI library and endpoints, scalar/packet channels for MCAPI library. We determine these objects based on the parallel pattern type and library used. Also, we use data sharing and thread communication characteristics of the original application to determine the objects. For example, when we are synthesizing MCAPI benchmarks, communication operations are message send/receive for geometric decomposition and recursive data patterns and packet/scalar send/receive for pipeline and event-based coordination patterns. Similarly, a task parallel benchmark uses barriers and a pipeline benchmark uses semaphores between stages.
- *Step* 2: Thread t_i performs initial computation operations before splitting the problem. For example, in divide and conquer parallel pattern, t_i splits the data into sub-partitions.
- *Step* 3: Thread t_i creates child threads if they exist for t_i in the original application. In some parallel patterns such as divide and conquer and recursive data, threads can be created dynamically by other threads during the execution.
- *Step* 4: Thread t_i gets the references of communication objects created by other worker threads at *Step* 1 (in case of Pthreads/MRAPI) or opens communication

channels (in case of MCAPI).

- *Step* 5: Parallel patterns except pipeline do not need to execute this step as they do not run in a loop.
- *Step* 6: Since threads need to access data before performing computation on the data, we add initial communication operations among threads according to the thread communication characteristics, thread communication matrix, of the original application. These operations are either read/write (in case of Pthreads/MRAPI) or message/packet send/receive operations (in case of MCAPI). We decide on the type of messages and the number of operations in this step. We also use mutex and semaphore objects in order to provide thread synchronization and ordering. For parallel patterns except task parallel, we add communication operations between threads that communicate in the original application.
- *Step* 7: After accessing the data either by reading shared memory or receiving message(s), t_i performs computation operations on this data, an increment in our case, and generates output data.
- *Step* 8: Thread t_i performs similar operations as done at *Step* 6 but this time the output data, which is generated at *Step* 7, is used during communication. For instance, in pipeline parallel pattern, t_i reads/receives the data from previous stage at *Step* 6, then processes the data at *Step* 7, and writes/sends the data to the next stage as seen in this step.
- *Step* 9: For pipeline parallel pattern, this is the end point of the loop, which begins at *Step* 5.
- *Step* 10: Thread t_i waits for child thread(s) to complete if they were created at *Step* 3.
- *Step* 11: If thread t_i has child thread(s) then it performs final computation operations after all threads are exited. For example, in divide and conquer parallel pattern, this is the operation phase after joining the worker threads.
- *Step* 12: Thread t_i deletes communication objects such as shared memory and semaphores created at *Step* 1. t_i also releases or closes communication objects, which are got or opened at *Step* 4.

The synthetic benchmark preserves data sharing characteristics of the original application by performing computation and communication operations described in the algorithm. For example, for the synthetic matrix multiplication example, the size of the global shared variables is calculated by using producer/consumer data sharing characteristic of the original application. Also, every thread performs equal number of iterations on one part of the shared data in a producer/consumer fashion according to the thread communication characteristics of the original application. Note that thread communication characteristics are preserved using *Step* 6 and *Step* 8. Also, we make sure that the synthetic preserves general threading characteristics as follows. The synthetic uses the same number of threads and each thread in the synthetic is created by the same thread as the original. Since we perform computation operations (*Step* 2, *Step* 7, *Step* 11) for each thread according to the lifetime relative to other threads, the synthetic preserves the lifetime and dynamic instruction count of each thread. Threads that run different functions in the original application run different functions in the synthetic.

Since our synthetics must have the same parallel pattern as the original, our tool first checks whether the parallel pattern of the synthetic is the same as the original. If it is not, the problematic group of characteristics is localized and then a synthetic with a reconfigured group of data sharing, thread communication, or general threading characteristics is generated until the parallel pattern matches or the number of iterations reaches the upper bound. For example, in order to improve thread communication characteristics, we either add the missing inter-thread communications between threads in the synthetic that exist in the original or remove the extra inter-thread communications that exist in the synthetic but not in the original. When we are improving thread communication similarity, we update operations at *Step* 6 and *Step* 8. Once the parallel pattern of the original and synthetic are the same, the synthetic preserves all 3 groups of high level characteristics including 13 sub-characteristics.

3.5.2. Similarity Measurement

After the first candidate synthetic benchmark with the correct parallel pattern is generated above, we use *similarity metrics* that are IPC, CMR, BMR, CCR to measure the similarity between the synthetic benchmark and the original application. Note that we do not use high level characteristics in similarity measurement because we make sure that the synthetic preserves these high level characteristics as described above. We use the individual similarity score to quantify the similarity of a synthetic benchmark and an original application. We calculate the overall similarity score as $oss_{cpu} = (iss_{IPC} + iss_{CMR} + iss_{BMR} + iss_{CCR})$ / 4.

3.5.3. Code Generation for Low Level Metrics

Once we know how to measure the similarity between the original application and the synthetic benchmark, at each iteration, we find the metric with the lowest individual score below the user defined threshold and improve the similarity for that score by adding code blocks suitable for that metric. The iterations continue until thresholds for individual and overall similarity scores are satisfied or the user defined threshold for the number of iterations is reached.

Figure 3.2 shows the C code blocks we use to increment/decrement IPC, CMR, and BMR. When iss_{IPC} is the lowest score, we either insert a C code block with high (integer addition) or low (division) IPC to the main function of the candidate synthetic benchmark. When iss_{CMR} is the lowest score and the CMR of the original is higher than the CMR of the synthetic, then we insert a code block that includes accesses to data that are not already cached. Otherwise, we insert a new code block where the data that are already in cache are accessed many times. When iss_{BMR} is the lowest score and the BMR of the original is higher than the BMR of the synthetic, then we insert a new code block with many branch mispredictions. Otherwise, we insert a new code block with many true branch predictions. In the case where iss_{CCR} is the lowest score and CCR of the original is higher than CCR of the synthetic, then we add communication operations to the synthetic. Otherwise, our code block contains

```
1   /** code block to increment IPC **/
2   for (i = 0; i < WORK_SIZE; i++) {
3     ires = i1 + i2; /* int i1, i2; */
4   }
5
6   /** code block to decrement IPC **/
7   for (i = 0; i < WORK_SIZE; i++) {
8     dres = d1 / d2; /* double d1, d2; */
9   }
10
11  /** code block to increment CMR **/
12  for (i = 0; i < WORK_SIZE; i++) {
13    array[rand() % arraySize] = 0; /* arraySize is larger than cache */
14  }
15
16  /** code block to decrement CMR **/
17  for (i = 0; i < WORK_SIZE; i++) {
18   for (a = 0; a < arraySize1; a++) { /* arraySize1 is smaller than cache */
19    for (b = 0; b < arraySize2; b++) {  /* arraySize2 is smaller than cache */
20      array[a][b] = 2 * array[a][b];
21  } } }
22
23  /** code block to increment BMR **/
24  for (i = 0; i < WORK_SIZE; i++) {
25    randNum = rand();
26    r3 = randNum % 3;
27    if (r3 == 0) { bres = b1 + b2 + (b1 / b2);  /* int b1, b2, bres; */ }
28    if (r3 == 1) { bres = b1 + b2 + (b1 / b2); }
29    r4 = randNum % 4;
30    if (r4 == 0) { bres = b1 + b2 + (b1 / b2); }
31    if (r4 == 1) { bres = b1 + b2 + (b1 / b2); }
32    r8 = randNum % 8;
33    if (r8 == 0) { bres = b1 + b2 + (b1 / b2); }
34    if (r8 == 1) { bres = b1 + b2 + (b1 / b2); }
35    if (r8 == 2) { bres = b1 + b2 + (b1 / b2); }
36    if (r8 == 3) { bres = b1 + b2 + (b1 / b2); }
37  }
38
39  /** code block to decrement BMR **/
40  for (i = 0; i < WORK_SIZE; i++) {
41    if (workCount >= 0) { /* always true prediction */
42      bres = b1 + b2 + (b1 / b2); /* int b1, b2, bres; */
43  } }
```

Figure 3.2. Code block to increment/decrement IPC, CMR, and BMR.

computations but not communication operations. Note that adding a new code block has side effects on other metrics. However, new code blocks do not affect the high level characteristics and do not change the parallel pattern of the synthetic. Once the appropriate code blocks are inserted into the synthetic, then we adjust $WORK_SIZE$ given in the figure according to the IPC of the original and synthetic. When the IPCs are close to each other, we use a small value for $WORK_SIZE$ and vice versa. Similarly, we adjust $WORK_SIZE$ for CMR and BMR.

3.5.4. A Detailed Example of CPU Benchmark Synthesis

We demonstrate our technique on a multi-threaded matrix multiplication application. We omit the original code and only show the synthetic in Figure 3.4. This application has geometric decomposition behavior where each matrix is divided into 3 parts and each worker thread works on one part. Note that we observe *producer/consumer* data sharing pattern and *many* thread communication pattern as expected in geometric decomposition. The general threading characteristics show that the *PCs* and *creator* of threads are the same because we have 3 threads created by the main thread and all threads execute the same function. Since each thread does multiplication operations on equal size data, *dynamic instruction counts* of the threads are similar. Also, since all threads are created at the beginning of the execution and their operation sizes are similar, we have the same *creation/exit times* as well as similar *lifetimes*.

Next, we recognize the parallel pattern of the original application. Figure 3.3 shows the parallel pattern recognition scores for data sharing, thread communication, and general threading characteristics in a Kiviat diagram. Since geometric decomposition has the highest score in total, our algorithm recognizes the parallel pattern correctly as geometric decomposition.

Then, we generate a miniaturized multicore synthetic benchmark for the matrix multiplication application using the algorithm given in Table 3.5. While generating this synthetic, we set the individual similarity score to 80 and overall similarity score to 90. Although the initial candidate benchmark preserves all high level characteristics, and

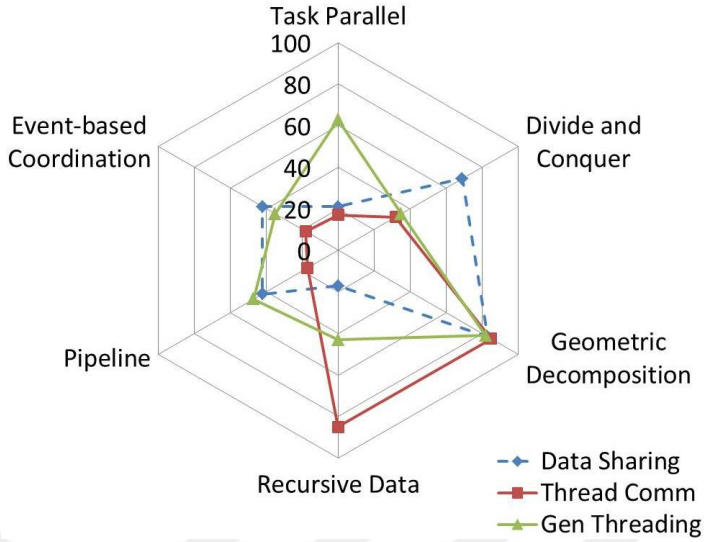

Figure 3.3. Parallel patterns scores of matrix multiplication.

the parallel pattern, it has different IPC, CMR, and BMR values which are 0.75, 0.30, and 0.40, respectively, whereas for the same performance characteristics the original has values 2.32, 0.17, and 0.55, respectively. The initial candidate synthetic benchmark has the following similarity scores: $iss_{IPC} = 33$, $iss_{CMR} = 24$, $iss_{BMR} = 73$, $iss_{CCR} = 93$, $oss_{cpu} = 56$. Since iss_{CMR} is the lowest score, we first add a code block to decrement CMR. In the following iterations, we add code blocks to increment IPC and to increment BMR. After 3 iterations we meet both the individual similarity scores and overall similarity score as follows: $iss_{IPC} = 92$, $iss_{CMR} = 94$, $iss_{BMR} = 91$, $iss_{CCR} = 94$, $oss_{cpu} = 93$. The IPC, CMR, and BMR values of the synthetic are 2.14, 0.18, and 0.50, respectively. We show the final synthetic in Figure 3.4. The execution times of the original and synthetic are 0.08 and 0.02 seconds, respectively. Hence, we have a 4× speedup. We achieve large speedups for large scale applications as will be shown in the experiments.

3.6. Synthetic Benchmark Generation for Embedded Multicore Systems

Embedded multicore systems are being deployed in many domains ranging from medical to automotive to networks. These embedded multicore systems may not be able to use traditional benchmarks such as PARSEC, Rodinia as well as our synthetic benchmarks in Pthreads. This is because these benchmarks rely on presence

```
1  typedef struct { unsigned int tid; } threadData;
2  int globAddr1[1200];  /* global memory: input */  // Step 1
3  int globAddr2[1200];  /* global memory: output */ // Step 1
4
5  void *task0(void *param) {
6    threadData* td = (threadData*) param;
7    int input, output, op; /* the variables used in Step 6, 7, 8 */
8    /* code block for worker thread 1 */
9    if (td->tid == 2) {
10     for (op = 0; op < 400; op++) { /* #operations decided in Step 7 */
11       input = globAddr1[op]; /* read from global mem */ // Step 6
12       output = input++; /* perform computation on input */    // Step 7
13       globAddr2[op] = output; /* write to global mem */ // Step 8
14   } }
15   if (td->tid == 3) {
16     for (op = 400; op < 800; op++) { /* #operations decided in Step 7 */
17       input = globAddr1[op]; /* read from global mem */ // Step 6
18       output = input++; /* perform computation on input */    // Step 7
19       globAddr2[op] = output; /* write to global mem */ // Step 8
20   } }
21   if (td->tid == 4) {
22     for (op = 800; op < 1200; op++) { /* #operations decided in Step 7 */
23       input = globAddr1[op]; /* read from global mem */ // Step 6
24       output = input++; /* perform computation on input */    // Step 7
25       globAddr2[op] = output; /* write to global mem */ // Step 8
26   } }
27   return NULL;
28 }
29
30 int main(int argc, char **argv) {
31   /* initializations */
32   threadData tData[3]; pthread_t threads[3]; int rc, t = 0, n;
33   /* code block for computation */
34   for(n = 0; n < 3; n++, t++) { /* Create and run all the threads */
35     rc = pthread_create(&threads[t], NULL, task0, (void *) &tData[t]);
36     if (rc) {
37       fprintf(stderr,``ERROR; return code from pthread_create() is %d\n'', rc);
38       exit(-1);
39   } }
40   for (t = 0; t < 3; t++) { /* Wait for all threads */
41     pthread_join(threads[t], NULL);
42   }
43   /* code blocks for CMR, IPC, and BMR to match similarity */
44   exit(0); /* global memory is removed automatically */ // Step 12
45 }
```

Figure 3.4. Synthetic matrix multiplication benchmark.

of shared memory architectures, or Pthreads, OpenMP, OpenCL libraries as well as uniform CPU ISAs where these embedded systems may not support such architectures or libraries. Hence, there is a need to develop benchmarks suitable for any given infrastructure, that is, SMP or message passing architectures, as well as benchmarks suitable for heterogeneous embedded multicore systems. In order to address above problems with benchmarks, we use the benchmark characteristics and software architectural patterns to develop synthetic benchmarks for embedded multicore systems suitable for any given infrastructure. Thanks to MCAPI and MRAPI, which are two of the standards developed by MCA [20], we target heterogeneous embedded multicore systems. MRAPI specifies essential application-level resource management capabilities to handle memory management and supply synchronization for multicore systems. MCAPI is a lightweight message passing API that aims to supply communication and synchronization between closely distributed embedded systems. Our framework also allows to change the communication paradigm between the original and synthetic. That is, if the original uses shared memory paradigm, the synthetic could use either shared memory (MRAPI) or message passing (MCAPI).

Our synthetic benchmarks are synthesized as C programs with MRAPI or MCAPI library and they preserve the microarchitecture independent and dependent behaviors. In order to achieve this, we measure the similarity between the original workload and the synthetic benchmark with respect to several similarity metrics. The *similarity metrics* that we use are the Parallel Pattern type (PL), Thread Communication (TC) behavior, Communication/Computation Ratio (CCR), IPC, CMR, and BMR. Many of the microarchitecture dependent metrics have previously been used to determine similarity but the software architectural patterns have not been used during synthesis.

We now describe how we calculate the individual similarity score for each metric.

Definition 3.8 (parallel pattern similarity score)**.** *The Parallel Pattern (PL) similarity score is calculated by comparing first whether the original and the synthetic have the same number of patterns, if not, a score of zero is generated. If they have the same number of patterns then we check the ratio of the number of matching pattern types in*

both workloads to the number of all pattern types in the original workload.

Definition 3.9 (thread communication similarity score)**.** *The Thread Communication (TC) similarity score is calculated by comparing the communication behavior between pairs of threads in the original workload and pairs of threads in the synthetic workload. We calculate it as follows:* $(CC + NN)/(CC + CN + NC + NN)$*. For a given pair of threads (assuming the number of threads is the same for both workloads), if the threads are communicating in both the original and the synthetic workloads, we increment the integer value* CC*. If the threads are not communicating in both the original and the synthetic workloads, we increment the integer value* NN*. If the threads are communicating in the original but not in the synthetic workload, we increment the integer value* CN*, and similarly we increment* NC*. This score gives us an accurate number in terms of the communication behavior.*

Definition 3.10 (communication to computation ratio similarity score)**.** *The Communication to Computation Ratio (CCR) similarity score is the average of the error rate of communication between the original and the synthetic workloads and the error rate of computation between the original and the synthetic workloads.*

Our synthesis flow makes sure that parallel pattern similarity score is always 100% for our synthetic benchmarks. The similarity scores for the remaining IPC, CMR, and BMR metrics are calculated by using individual similarity score formula. Finally, we calculate the overall similarity score as $oss_{emb} = (iss_{TC} + iss_{CCR} + iss_{IPC} + iss_{CMR} + iss_{BMR})$ / 5. We do not use the PL score because we make sure that the synthetic and the original workloads have the same types of patterns.

We follow the steps given in Table 3.5 to generate synthetic embedded multicore benchmarks. When all the steps given above are completed, a miniaturized multicore synthetic benchmark for embedded multicore systems is obtained. This synthetic benchmark keeps the performance attributes of the original workload as we show in the experimental works. Note that our framework allows changing the communication paradigm between the original and synthetic. That is, if the original uses shared memory paradigm, the synthetic could use either shared memory (MRAPI) or message

Table 3.6. Multicore machine configurations.

Parameter	System-I	System-II	System-III	System-IV
#Cores	4	2x4	8	2
#Logical procs	8	2x8	8	4
DRAM	6 GB	32 GB	4 GB	4 GB
L1 I/D	32 KB, 8 way	32 KB, 4 way	32 KB, 8 way	32 KB, 8 way
L2	256 KB, 8 way	256 KB, 8 way	128 KB, 8 way	256 KB, 8 way
LLC (L3)	6 MB, 12 way	8 MB, 16 way	2 MB, 32 way	3 MB, 12 way
Branch Predictor	2-level correl, 64-bit	2-level correl, 64-bit	512-entry, 2-bit	2-level correl, 64-bit
Architecture	x86, 64-bit Core i7	x86, 64-bit Xeon e5520	Power, 64-bit FSL, P4080ds	x86, 64-bit Core i5

passing (MCAPI).

3.7. Experiments

We performed experiments to validate our benchmark characterization, pattern recognition, and benchmark synthesis techniques. In order to show that our approach works across different architectures and different number of cores and cache sizes, we targeted 4 different actual multicore machines as shown in Table 3.6. During experiments, Intel SpeedStep® and Hyper-Threading technologies were enabled and threads were not pinned to cores. We used GCC 4.6.1 for x86_64 Ubuntu Linux on System-I, System-II, and System-IV and GCC 4.6.2 for P4080ds Linux on System-III. We compiled original benchmarks with default options, and synthetic benchmarks with '-O0' option so that the compiler did not remove our code blocks.

We used PARSEC [3], Rodinia (OpenMP) [4], and EEMBC MultiBench [6] benchmarks as original benchmarks and generated synthetic benchmarks that use Pthreads, MRAPI, or MCAPI libraries. These benchmarks cover a big range of multicore benchmarks that are available. We use EEMBC MultiBench benchmarks that are multi-threaded and that have a single kernel. We used the test input for PARSEC, default

input for Rodinia, and medium input for EEMBC MultiBench.

We implemented our techniques in MINIME tool that consists of nearly 10K lines of C code and 500 lines of Python script. Our tool and all of our benchmarks can be downloaded from our website[1] . We ran the original and synthetic benchmarks 10 times in order to obtain similarity scores. We also set the maximum number of iterations to 40, the overall similarity score to 90%, and individual similarity scores to 80%. The similarity scores we used are the maximum achievable scores with our framework. We display results for synthetic benchmarks that use Pthreads library but we generated synthetic benchmarks for MRAPI and MCAPI libraries as well and had similar results. Also, due to compilation and binary instrumentation problems, we do not list results for all applications in these benchmark suites.

We generated two sets of synthetics, first on x86 ISAs (specifically on System-I) then on Power ISA. This is because each ISA has different characteristics and constraints that impact the behavior of a benchmark such as stack operations, ISA-specific complex operators, and calling conventions. Furthermore, our high level characterization tools DynamoRIO and Umbra currently support x86 ISA, hence we devised another technique to generate the synthetic on Power ISA. First, we generate the synthetic with high level characteristics on System-I. We then start from this synthetic on System-III and add code blocks for low level characteristics on System-III. Note that the high level structure of the synthetic benchmark does not change going from x86 to Power ISA.

3.7.1. Evaluation of Benchmark Synthesis

Table 3.7 shows the results of our pattern recognition and synthesis results on System-I. In the table, we show the *parallel pattern* of the original benchmark found by us through code analysis, the lines of code (LOC), and the number of iterations ($\#iter$) it takes to generate the synthetic benchmark. We also validated the parallel patterns of PARSEC benchmarks from the literature since they are available. Note that

[1] http://depend.cmpe.boun.edu.tr/tools/minime

Table 3.7. Pattern recognition and synthesis results.

Original Benchmark				Synthetic Benchmark				
Suite	**Benchmark**	**LOC**	**Parallel Pattern**	**LOC**	**#iter**	**Speedup(×)**	**CodeSize(×)**	**OSS**
PARSEC	Blackscholes	1262	Task Parallel	124	2	10	10	95
	Bodytrack	7696	Geometric Decomposition	403	16	11	19	90
	Canneal	2794	Task Parallel	136	2	22	20	93
	Dedup	7125	Pipeline	440	9	36	16	94
	Facesim	20275	Task Parallel	127	5	15	159	94
	Ferret	10765	Pipeline	1426	5	67	7	90
	Fluidanimate	2784	Geometric Decomposition	330	2	15	8	90
	Swaptions	1095	Task Parallel	144	2	13	7	91
	X264	38546	Pipeline	940	12	26	41	92
Rodinia	Kmeans	2146	Task Parallel	180	15	36	11	90
	HotSpot	196	Geometric Decomposition	195	10	16	1	94
	Back Propagation	478	Task Parallel	129	5	20	3	94
	SRAD	495	Task Parallel	222	17	12	2	93
	Breadth-First Search	125	Task Parallel	195	18	35	0	94
	CFD Solver	1539	Task Parallel	243	11	10174	6	90
	LU Decomposition	541	Geometric Decomposition	199	32	10	2	94
	Heart Wall Tracking	2244	Task Parallel	180	16	34	12	90
	Particle Filter	398	Geometric Decomposition	242	9	10	1	91
	PathFinder	127	Geometric Decomposition	343	14	13	0	95
	LavaMD	353	Geometric Decomposition	274	19	30	1	90
EEMBC MultiBench	idctrn01	655	Task Parallel	284	20	16	2	94
	md5	188	Geometric Decomposition	332	10	10	0	94
	ippktcheck	693	Geometric Decomposition	362	11	10	1	94
	ipres	1508	Geometric Decomposition	343	36	42	4	90
	rotatev2	804	Task Parallel	222	5	10	3	90
	mp2decode	9089	Geometric Decomposition	622	12	11	14	93

parallel patterns of Rodinia benchmarks are not known from the literature. However, our framework finds that Rodinia benchmarks have only task parallel and geometric decomposition patterns. This is expected because OpenMP does not support other patterns. That is, if the data used in OpenMP is private, then it results in task parallel pattern, otherwise the pattern is geometric decomposition. We observe that PARSEC and Rodinia benchmark suites do not contain all parallel patterns such as recursive data pattern. The column *Speedup(×)* shows speedup obtained in terms of execution time and the column *CodeSize(×)* refers to the reduction in lines of code going from the original to the synthetic.

When generating synthetics, we make sure that they have exactly the same parallel patterns as the original benchmarks. Our synthetics have the same number of threads as the originals, hence they are not generated for a specific number of cores. Recognizing and generating the parallel pattern correctly is the most important step in a synthetic because recognizing a wrong pattern can result in wrong communication and computation behaviors as well as dissimilar performance characteristics in the synthetic. This can also result in higher number of iterations to match the synthetic with the original one or not be able to match at all. For example, when we manually force the parallel pattern of `Ferret` benchmark from PARSEC as task parallel instead of pipeline, we obtain a synthetic with only 50% overall similarity score even after 20 iterations. However, our pattern recognizer correctly recognizes the parallel pattern as pipeline and our synthesizer generates a synthetic in 5 iterations with 90% overall similarity score. This observation explicitly indicates a relationship between the high level architectural pattern and performance characteristics, which we will experimentally show later as well.

From the table, it can also be seen that the synthetics are much smaller and faster hence less complex than the originals leading to high simulation speeds, which is one of the main goals of this study. The average speedup is 21× (without `CFD Solver`) and the average code reduction is 14×. Note that *CodeSize(×)* is denoted as 0 in some cases. This corresponds to the cases where the original code size is very small hence the synthetic code size is larger than the original, yet the synthetic can run much faster. The execution time of `CFD Solver` benchmark from Rodinia is 305.22 seconds, which is the longest among all benchmarks and the execution time of the synthetic is 0.03 seconds. Hence, we have 10174× speedup. It is clear that when the execution time or the code size of an original benchmark is high, we have a larger speedup or code size reduction.

In most cases, we generate the synthetic after 20 iterations. When the value of any characteristic of an original benchmark is too low or too high, the $WORK_SIZE$ of the code block we add to the synthetic becomes larger. This results in high influence on other characteristics that we try to match and managing these side effects requires more

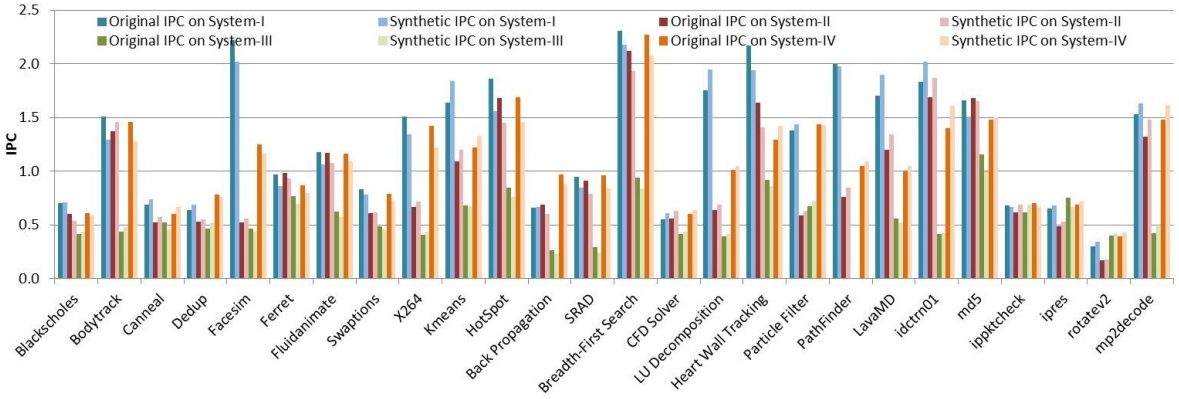

Figure 3.5. Comparison of IPC between the synthetic and original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III.

iterations and may result in lower similarity scores. For example, `LU Decomposition` took 32 iterations because the cache miss rate of the original benchmark is very low. Similarly, `ipres` took 36 iterations because the BMR of the original benchmark is very high. Also, the execution time of the synthetic benchmark increases with the increasing iteration size.

In the table, we show the overall similarity scores in the column *OSS* for System-I where the average is 92%. The average overall similarity scores are 91%, 92%, and 90% for System-II, System-III, and System-IV, respectively. These scores show that synthetics are above the range set by the user (90%) and have high degree of similarity with the originals.

3.7.2. Assessing Similarity

We next compare the similarity of our synthetic benchmarks with the original benchmarks in order to validate the accuracy of our synthetic benchmarks where we use the similarity metrics described in Section 3.5, which are IPC, CMR, BMR, and CCR. We calculated the error between the synthetic benchmark and the original benchmark with respect to each of these metrics. We also present the average and maximum error for each metric.

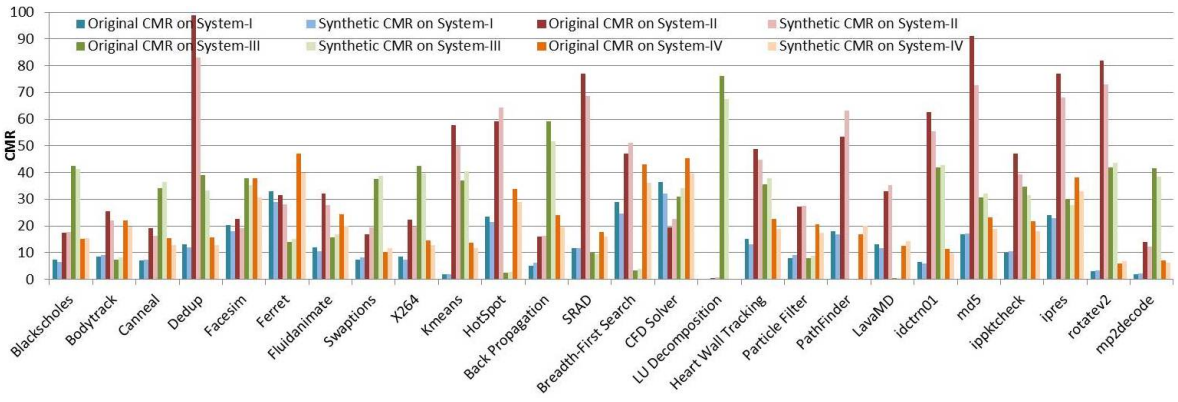

Figure 3.6. Comparison of CMR between the synthetic and the original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III.

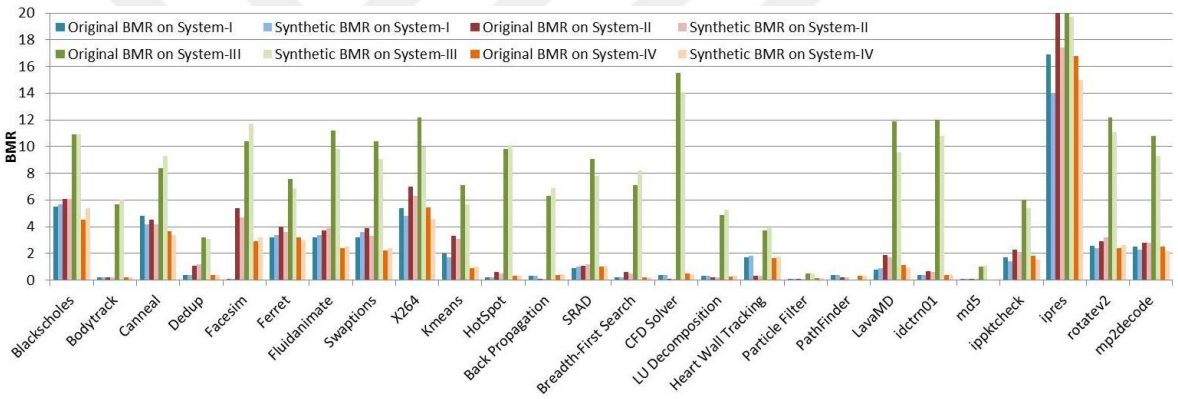

Figure 3.7. Comparison of BMR between the synthetic and the original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III.

Figures 3.5, 3.6, and 3.7 compare IPC, CMR, and BMR between the synthetic and original benchmarks, respectively, for the four systems. The average errors on System-I are 8%, 10%, 6%, and 8% and the maximum errors are 16%, 17%, 18%, 17% for IPC, CMR, BMR, and CCR, respectively. We also measured the similarity scores for Instruction and Data Level 1 cache hit rates on System-I and the average errors are 1% and 5%, respectively. We obtain similar results on other systems where the average errors for Instruction and Data Level 1 cache hit rate are smaller than 2% and 7%, respectively. The results show that all synthetics are similar to the originals within the bounds set by the user and are acceptable for a high level synthetic benchmark.

We observe that improving one individual score can worsen other scores, that is, the added code block can have side effects. For example, at iteration 10 of `ippktcheck`, iss_{CMR} and iss_{BMR} are 65 and 90, respectively. Hence, we add a code block to improve CMR score. However, after addition of this code block at iteration 11, iss_{CMR} and iss_{BMR} become 84 and 82, respectively. That is, BMR score is decreased. Although this show that the addition of code blocks can have side effects, using our algorithm given in Table 3.5, synthetics have been successfully generated for the given similarity scores. If the user wants to obtain synthetics with much higher similarity, match instruction mix, or match Cycles Per Instruction (CPI) stack, we can use inline assembly in our code blocks as an alternative technique. However, this will lead to synthetics that are not portable, hence we chose not to follow this technique.

3.7.3. Assessing Architecture Changes

We next compare hardware configuration independence (portability) of our synthetics by first generating the synthetic on System-I and then executing the same synthetic on System-II and System-IV. Note that we could not use this synthetic on System-III as it has a different ISA as described above. When we used the synthetic generated on System-I on System-III, high level characteristics of this synthetic such as parallel pattern matched but low level characteristics did not match as expected.

Figure 3.5 compares the IPC between the synthetic and original benchmarks on four systems, where the synthetic generated on System-I is used on System-II and System-IV. We see that when the IPC of the original benchmark changes from System-I to System-II and System-IV, the IPC of the synthetic benchmark changes similarly, which is what we want for portability. The average IPC error on four systems is 8%, 9%, 9%, and 9% and the maximum IPC error is 16%, 14%, 16%, and 16%, respectively. For example, the IPC of `Kmeans` benchmark is 1.64, 1.09, and 1.22 on System-I, System-II, and System-IV, while the IPC of the synthetic benchmark is 1.84, 1.20, and 1.33, respectively. The average CMR error on four systems is 10%, 10%, 8%, and 10% and the maximum CMR error is 17%, 20%, 15%, and 17%, respectively. The average BMR error on four systems is 6%, 7%, 10%, and 6% and the maximum BMR error is 18%,

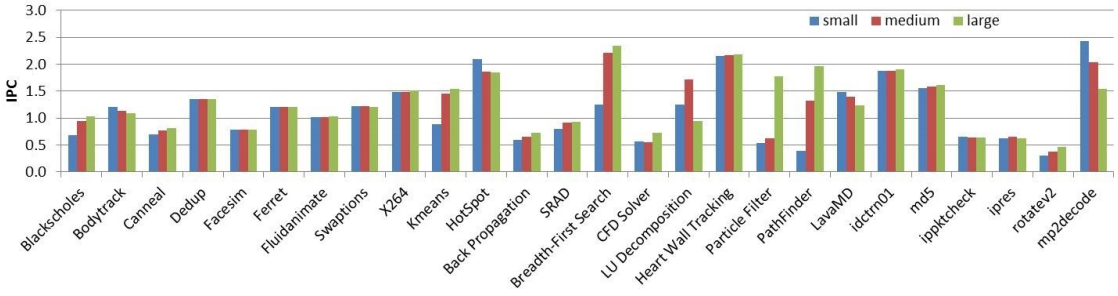

Figure 3.8. IPC values of original benchmarks for small, medium, and large inputs on System-I.

17%, 20%, and 18%, respectively. We obtain similar results for CCR and Instruction and Data Level 1 cache hit rates. Hence, our synthetics are portable across different hardware configurations.

3.7.4. Assessing Input Changes

We analyze the impact of input changes on characteristics of original and synthetic benchmarks. In particular, we test whether we can use the synthetic that is generated for a particular input size, say small, as a synthetic for other input sizes, say medium or large. To test this claim, one can check whether the individual and overall similarity scores are met, when the synthetic for a particular input size is used for other input sizes. For this purpose, we run original benchmarks from PARSEC, Rodinia, and EEMBC MultiBench with small, medium, and large inputs.

We observed that when input sizes change, individual similarity scores for high level characteristics are met for all benchmarks but individual similarity scores for low level characteristics are not met for some benchmarks. In particular, when we consider all low level characteristics, we observe that for 18 of the 26 benchmarks, a single synthetic for one input type can be used for other input sizes. We display the IPC values of original benchmarks for small, medium, and large inputs on System-I in Figure 3.8. From the figure, we see that for `ipres` benchmark, IPC values remain nearly the same for different inputs (0.63, 0.65, and 0.63 for small, medium, and large inputs, respectively). This is also the case for other low level characteristics of this benchmark.

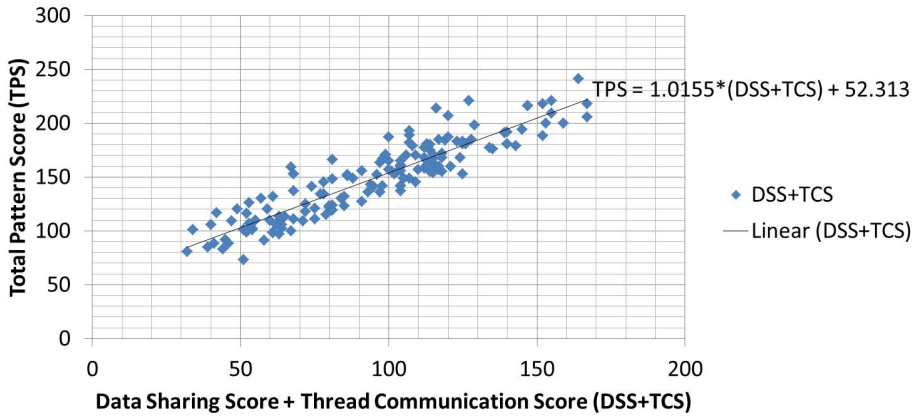

Figure 3.9. Linear regression analysis between Data Sharing Score + Thread Communication Score and Total Pattern Score.

Hence, we can use the same synthetic generated for one input size for different input sizes. Whereas, for `Blackscholes` benchmark, IPC values change drastically (0.68, 0.95, and 1.03 for small, medium, and large inputs, respectively). Hence, we cannot use the synthetic, say for the small input size for other input sizes, as the similarity scores do not match.

3.7.5. Correlation between Parallel Pattern Score and Overall Similarity Score

We ran regressions to analyze the correlation between high level characteristics and the parallel pattern. We found that the combination of data sharing and thread communication characteristics has the highest correlation with the parallel pattern. Figure 3.9 shows this relationship where the correlation coefficient is 0.91.

Next, we analyze the correlation between the parallel pattern score and the overall similarity score. In order to check this correlation, from a given original application, we generated six synthetic benchmarks each using a different parallel pattern and only one having the correct pattern. Note that these synthetics are obtained only after one iteration step and no code blocks have been added for matching similarity metrics. We then calculate the total pattern score and overall similarity score for each synthetic. We observed that when the synthetic gets the highest total pattern score, that is, when

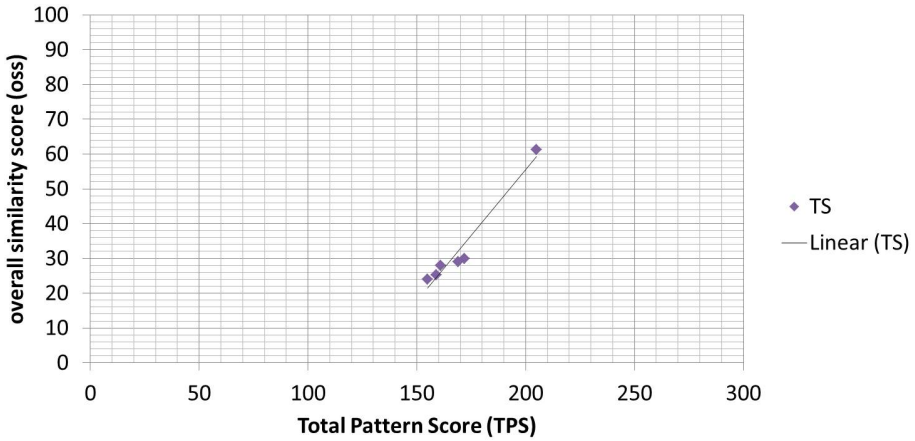

Figure 3.10. Linear regression analysis of `Bodytrack` for parallel pattern and overall similarity score relation.

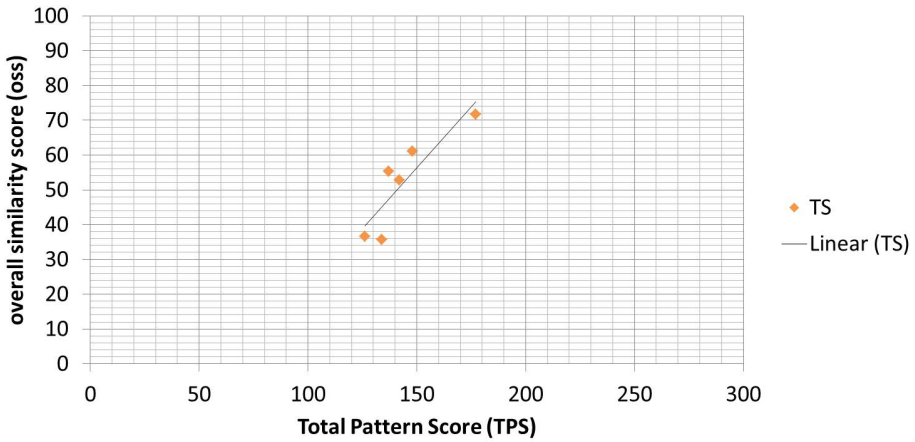

Figure 3.11. Linear regression analysis of `ippktcheck` for parallel pattern and overall similarity score relation.

it has the correct parallel pattern, we also have the highest overall similarity score. Figures 3.10 and 3.11 show a linear correlation between the total pattern score and the overall similarity score for `Bodytrack` and `ippktcheck` benchmarks, respectively. In the figure, the overall similarity score for `Bodytrack` is 61, when it has the correct parallel pattern and it is between 24 and 30 for other parallel patterns. Similarly, the overall similarity score for `ippktcheck` is 72 for the correct parallel pattern and between 36 and 61 for other parallel patterns. Moreover, there exists a linear correlation between the total pattern score and the overall similarity score. Correlation coefficient for `Bodytrack` is 0.96 and it is 0.88 for `ippktcheck`. The average correlation coefficient is 0.68 for all benchmarks used during experiments.

Table 3.8. Multicore hardware configurations.

Parameter	**HW1**	**HW2**
#Cores	4	2x4
#Logical procs	8	2x8
DRAM	6 GB	32 GB
L1 I/D	32 KB, 8 way	32 KB, 4 way
L2	256 KB, 8 way	256 KB, 8 way
LLC (L3)	6 MB, 12 way	8 MB, 16 way
Branch Predictor	2-level correl, 64-bit	2-level correl, 64-bit
Architecture	x86, 64-bit Core i7	x86, 64-bit Xeon e5520

3.7.6. Synthetic Benchmark Generation for Embedded Multicore Systems

We performed experiments to analyze the correlation (similarity) of synthetic benchmarks for embedded multicore systems and real (original) benchmarks. In order to show that our approach works across different number of multicores and cache sizes, we targeted different core and cache platforms. The experiments were performed on two hardware configurations as shown in Table 3.8. We used PARSEC [3], and Rodinia (OpenMP) [4] benchmarks as real benchmarks and generated synthetic benchmarks in MRAPI and MCAPI from them. We ran the original and the synthetic benchmarks 10 times in order to obtain similarity scores. We also set the number of iterations to 20, the overall similarity score to 80% and individual similarity scores to 70%. We used the medium input for PARSEC and default input for Rodinia.

Table 3.9 shows our benchmark synthesis results. We show the lines of code (LOC) for the original and the synthetic benchmarks as well as the number of iterations (#iter) it takes to generate the synthetic benchmark. It can be seen that the synthetic is much smaller and less complex than the original as expected, hence leading to high simulation speeds. Also, in general, we generate the synthetic after only a few iterations. `X264` took 17 iterations because the synthetic benchmark is large in

Table 3.9. Benchmark synthesis results for embedded multicore systems.

Original			Synthetic	
Suite	**Benchmark**	**LOC**	**LOC**	**#iter**
PARSEC	Blackscholes	1262	116	1
	Bodytrack	7696	1197	5
	Canneal	2794	116	1
	Dedup	7125	756	1
	Facesim	20275	190	1
	Ferret	10765	2722	4
	Fluidanimate	2784	867	9
	Swaptions	1095	189	6
	X264	38546	1647	17
Rodinia	Kmeans	2146	177	9
	HotSpot	196	130	3
	Back Propagation	478	130	15
	SRAD	495	115	6
	Breadth-First Search	125	185	8
	CFD Solver	1539	189	1
	LU Decomposition	541	553	5
	Heart Wall Tracking	2244	177	4
	Particle Filter	398	189	14
	PathFinder	127	190	2
	LavaMD	353	190	4

terms of lines of code as well as the number of library function calls. These result in high influence on the metrics that we are trying to match. Furthermore, `X264` has two patterns that makes it harder to synthesize.

We next compared the similarity of our synthetic benchmarks with the real benchmarks using both microarchitecture independent metrics such as PL, TC, and CCR as well as microarchitecture dependent metrics such as IPC, (L1 and L2) CMR, and BMR.

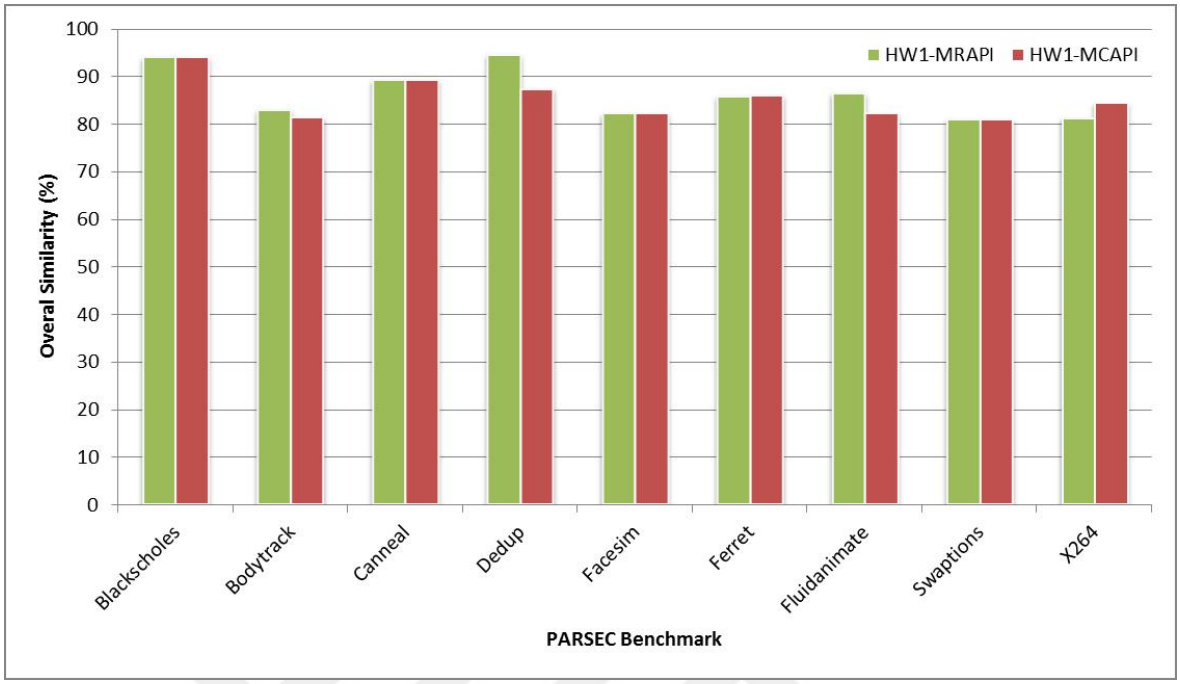

Figure 3.12. Overall similarity scores of synthetic benchmarks from PARSEC for MRAPI/MCAPI on HW1.

We calculated the error between the synthetic benchmark and the original benchmark with respect to each of these metrics. We also present the average error for each metric. The first set of experiments are performed on hardware configuration HW1.

Figures 3.12 and 3.13 compare the overall similarity scores of the synthetic benchmarks for MRAPI and MCAPI on HW1. The average similarity score is 87% and the minimum similarity score is 81% for MRAPI in `Swaptions`, `x264`, and `Heart Wall Tracking`. The maximum similarity score for MRAPI is `Dedup` with 95%. The average similarity score is 86% and the minimum similarity score is 81% for MCAPI in `Swaptions`, `Bodytrack`, and `Heart Wall Tracking`. The maximum similarity score for MCAPI is 94% for `Blacksholes` and `Back Propagation`. We observe that the synthetic and the original workloads are similar to each other over 80%, which was what was set by the user. These scores also show the high quality of synthetics.

Figures 3.14 and 3.15 compare Thread Communication score of the synthetic benchmarks for MRAPI and MCAPI. The average error is 4% and the maximum error is 17% for MRAPI in `Bodytrack`. The average error is 2% and the maximum

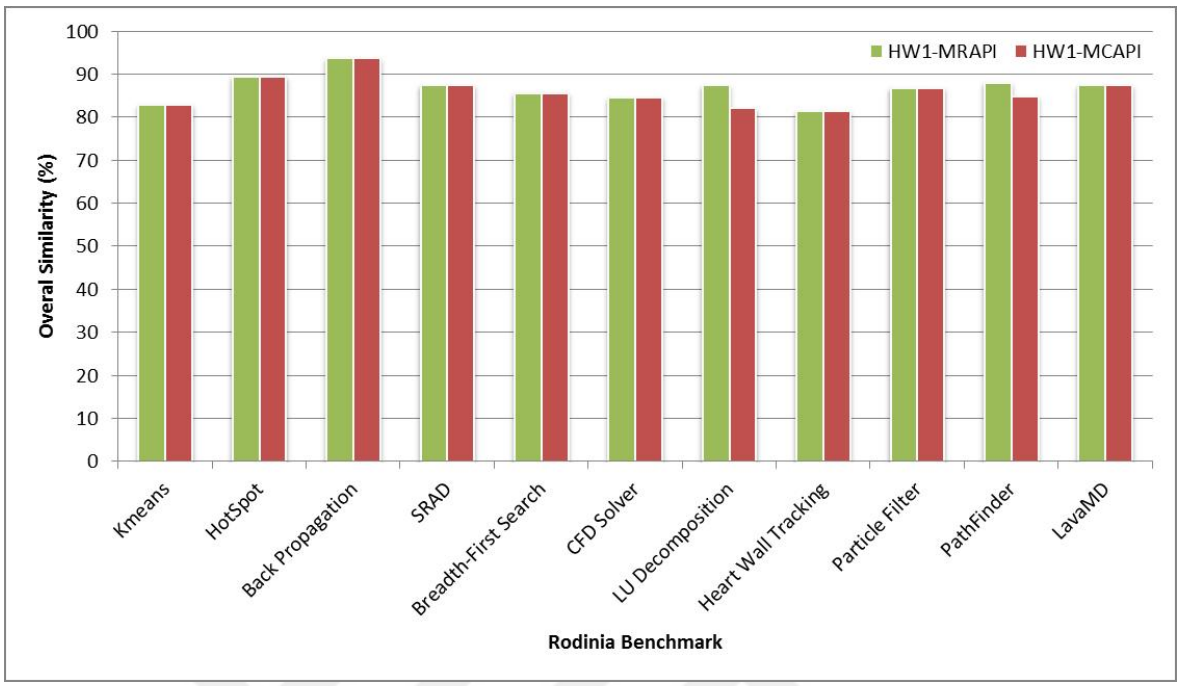

Figure 3.13. Overall similarity scores of synthetic benchmarks from Rodinia for MRAPI/MCAPI on HW1.

error is 10% for MCAPI in `Canneal`. MRAPI and MCAPI overall similarity and thread communication scores are close to each other. They both use the same library platform hence this is expected. Unless specified otherwise, we display results for MRAPI synthetic benchmarks.

Figures 3.16 and 3.17 compare CCR between the synthetic and the real benchmarks for MRAPI. The average error is 19% and the maximum error is 29% for `Kmeans`. We observe that improving one metric can worsen others. Specifically, for `Kmeans`, we added a C code block in order to decrease the cache miss rate, which led to an increase in CCR error. Figures 3.18 and 3.19 compare IPC between the synthetic and the real benchmarks. The average error is 16% and the maximum error is 30% for `Particle Filter`. Figures 3.20 and 3.21 compare CMR between the synthetic and the real benchmarks. The average error is 16% and the maximum error is 30% for `Ferret`, and `LU Decomposition`. The reason why these synthetic benchmarks have large error is that the real benchmarks have the smallest (0.2%) and the highest (33.7%) cache miss rates that result in high loop counts with side effects in our synthetics. Figures 3.22 and 3.23 compare BMR between the synthetic and the real benchmarks. The average

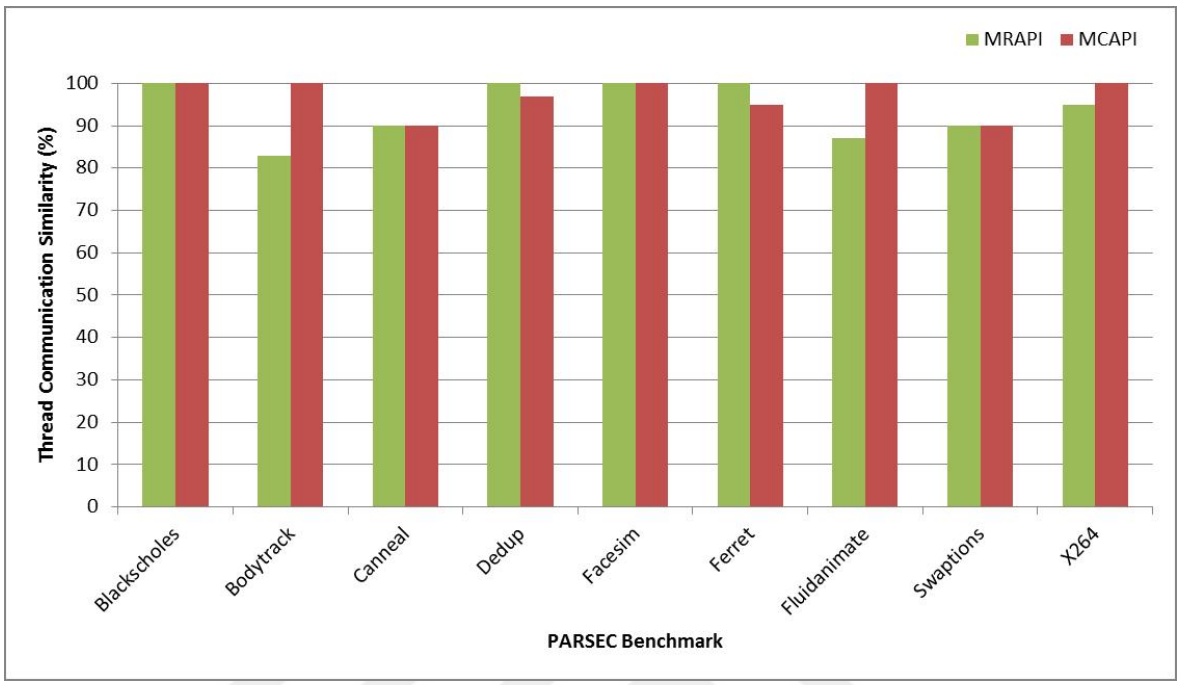

Figure 3.14. Thread Communication scores of the synthetic benchmarks from PARSEC for MRAPI and MCAPI.

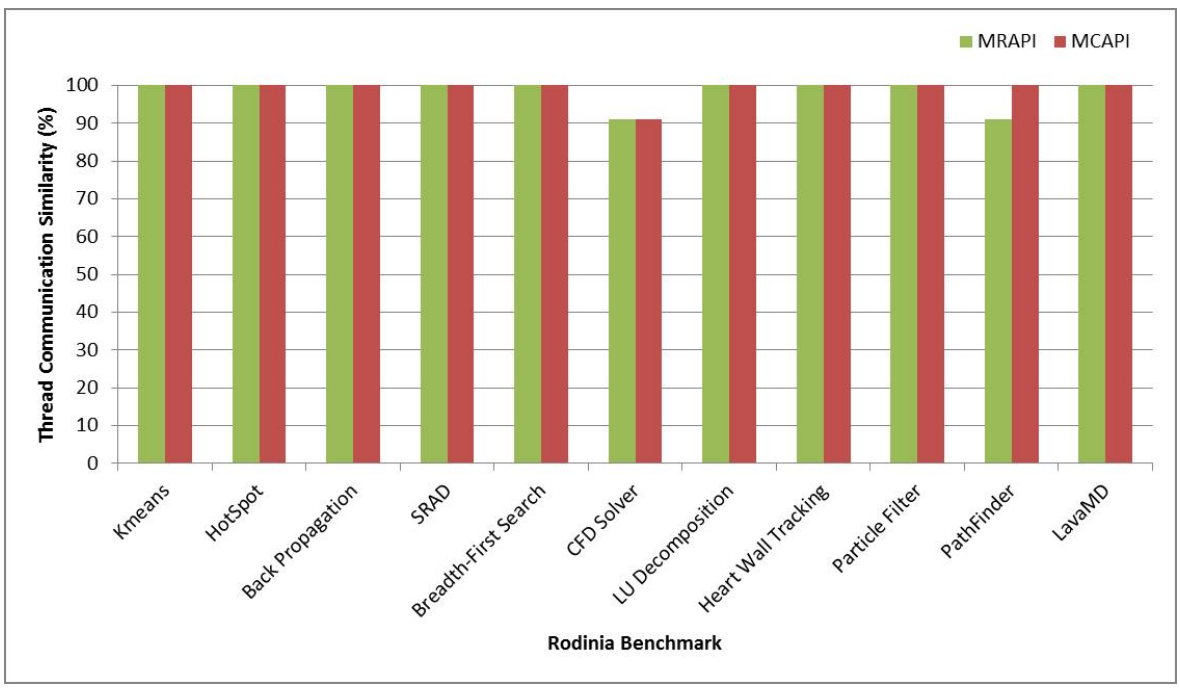

Figure 3.15. Thread Communication scores of the synthetic benchmarks from Rodinia for MRAPI and MCAPI.

error is 12% and the maximum error is 29% for `X264`. Note that the average error for the above set of metrics is 16% and the maximum error is 30%. This is expected since our goal is to maximize those high level metrics such as the parallel pattern and thread communication. Even though these error results may seem high the overall score is still above 85% on average.

We also performed experiments where we increased the user defined overall similarity scores to 90%. We observed that the lines of code in the synthetic benchmarks do not increase however the number of iterations goes up. Also, we are unable to reach 90% for benchmarks where the microarchitecture dependent metrics such as cache miss rate is very low. However, there is a lot of work in the literature that develops synthetics with these low level metrics and we plan to exploit those works in the future.

We next compare hardware configuration independence of our results by running experiments on HW2. All of the runs on HW2 use the same synthetic benchmarks synthesized from the HW1 configuration, not re-synthesized benchmarks. Figures 3.24 and 3.25 compare the overall similarity scores of the synthetic benchmarks for MRAPI and MCAPI on HW2. The average similarity score is 85% and the minimum similarity score is 81% for MRAPI in `Swaptions`, `X264`, and `Heart Wall Tracking`. The maximum similarity score for MRAPI is 93% for `Dedup` and `Back Propagation`. The average similarity score is 84% and the minimum similarity score is 81% for MCAPI in `Swaptions` and `Heart Wall Tracking`. The maximum similarity score for MCAPI is 93% for `Back Propagation`. Figures 3.26 and 3.27 compare IPC between the synthetic and the real benchmarks on HW2. The average error rate is 16% and the maximum error rate is 30% for `Particle Filter`. We observe from HW2 results that the both the overall similarity scores and IPC scores are independent of hardware configurations. In other words, we observe nearly the same scores on both hardware configurations.

3.7.7. Discussion

Our techniques allow designers to use synthetic benchmarks in the early design stage of multicore systems where benchmarks need to be run frequently on hardware

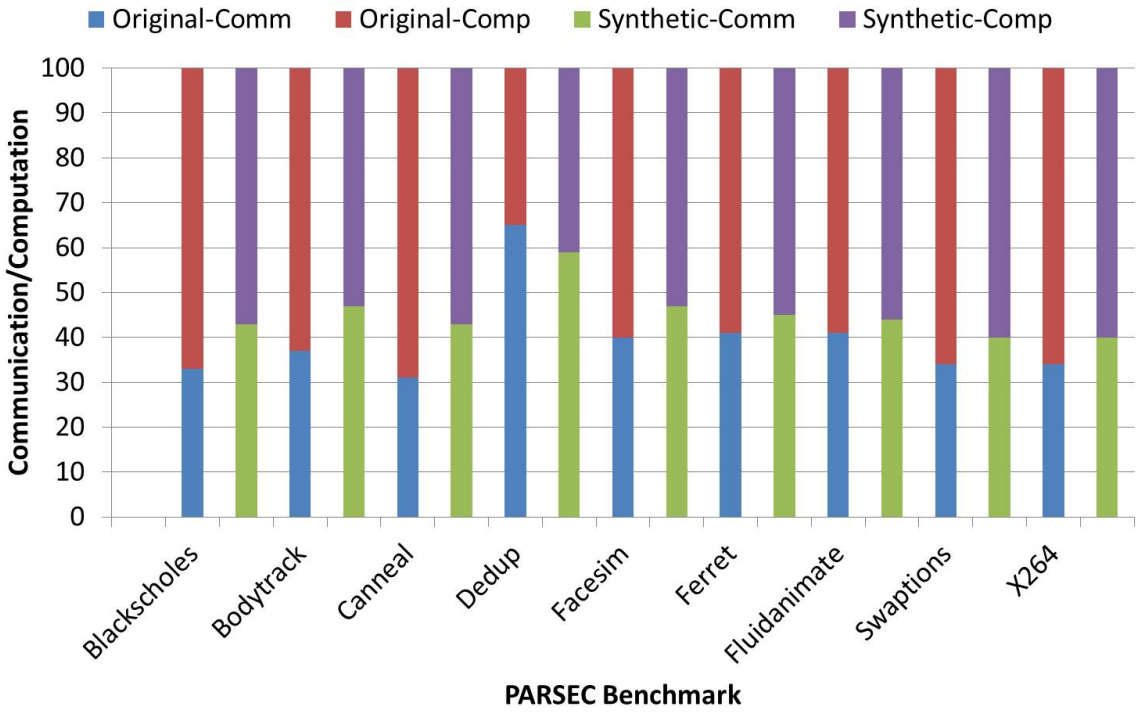

Figure 3.16. Comparison of CCR between the synthetic and the original benchmarks from PARSEC.

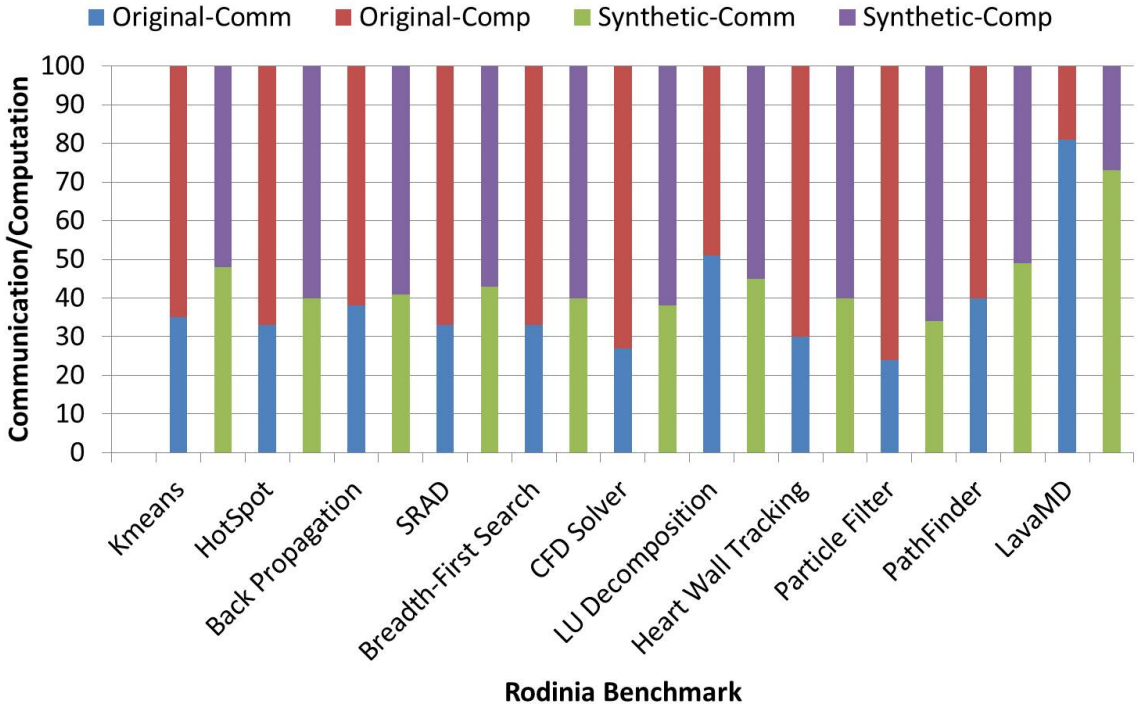

Figure 3.17. Comparison of CCR between the synthetic and the original benchmarks from Rodinia.

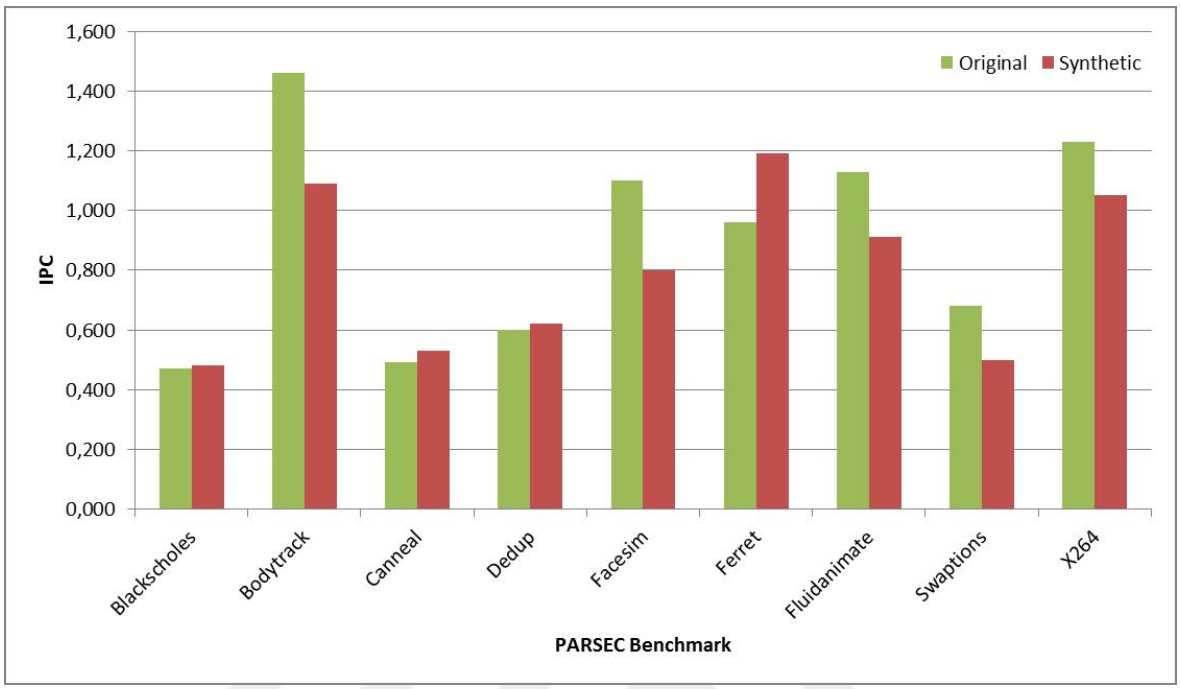

Figure 3.18. Comparison of IPC between the synthetic and the original benchmarks from PARSEC for MRAPI.

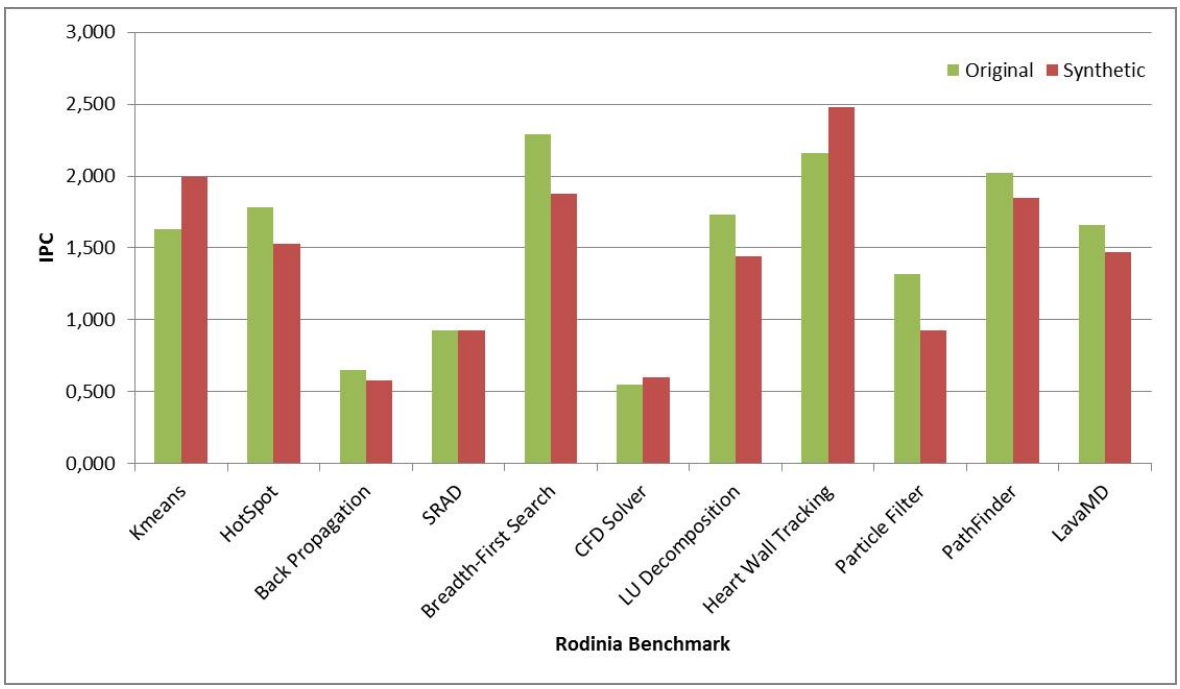

Figure 3.19. Comparison of IPC between the synthetic and the original benchmarks from Rodinia for MRAPI.

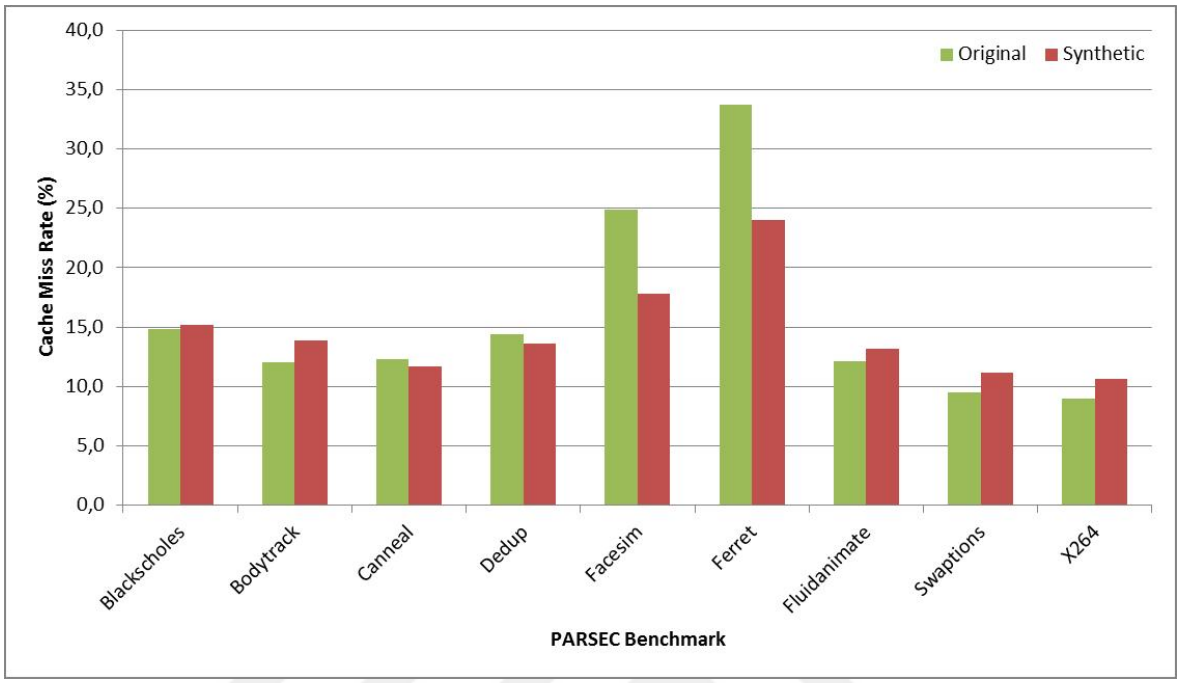

Figure 3.20. Comparison of CMR between the synthetic and the original benchmarks from PARSEC.

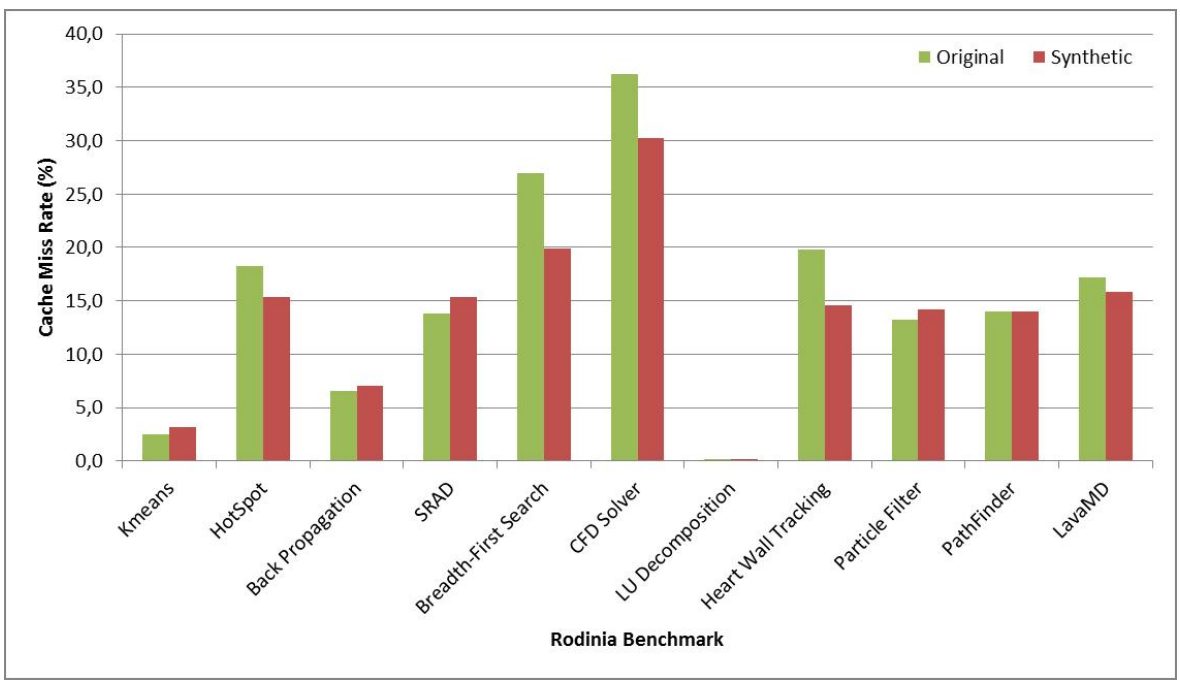

Figure 3.21. Comparison of CMR between the synthetic and the original benchmarks from Rodinia.

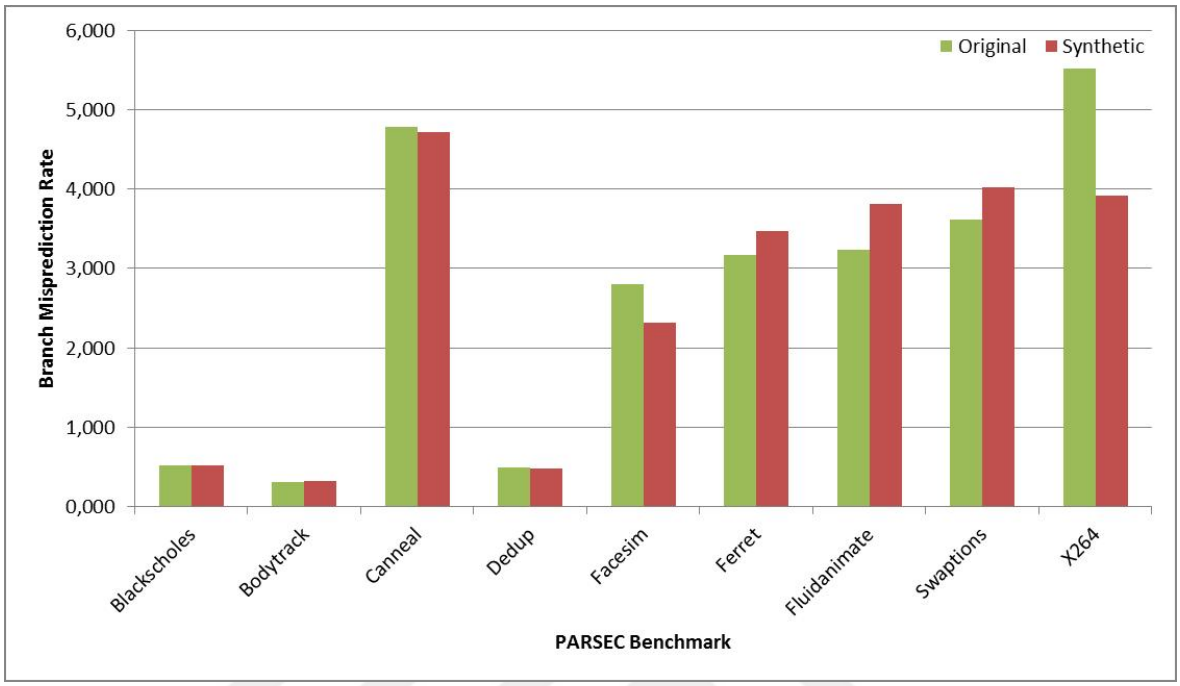

Figure 3.22. Comparison of BMR between the synthetic and the original benchmarks from PARSEC.

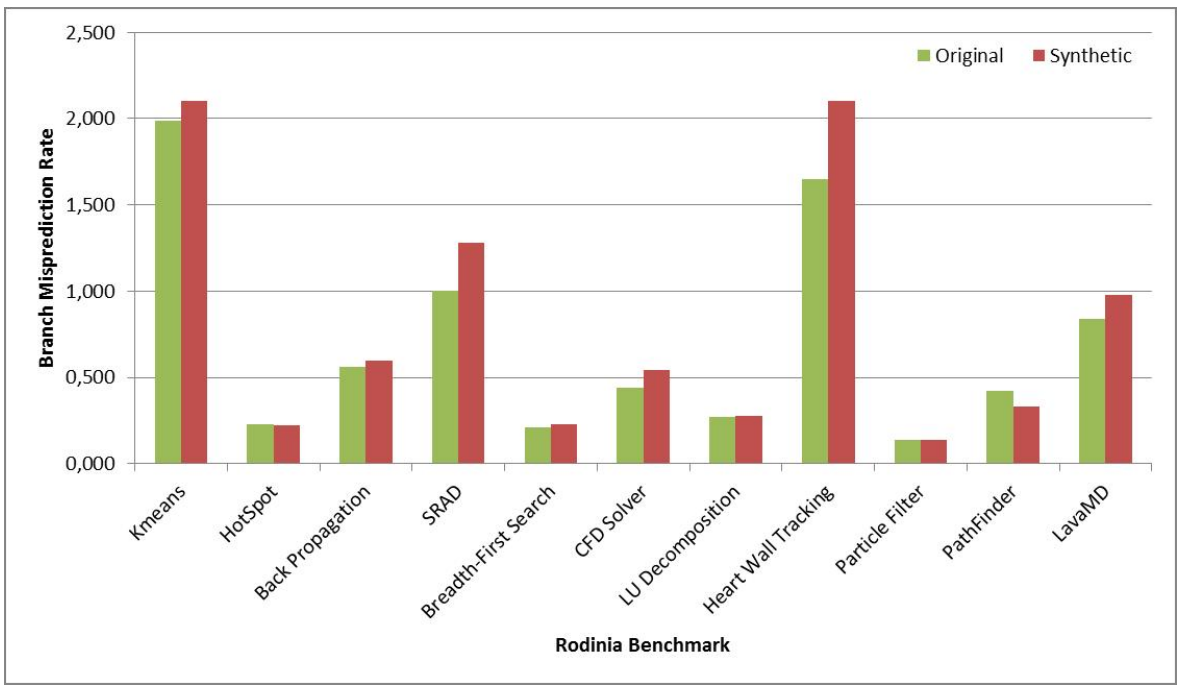

Figure 3.23. Comparison of BMR between the synthetic and the original benchmarks from Rodinia.

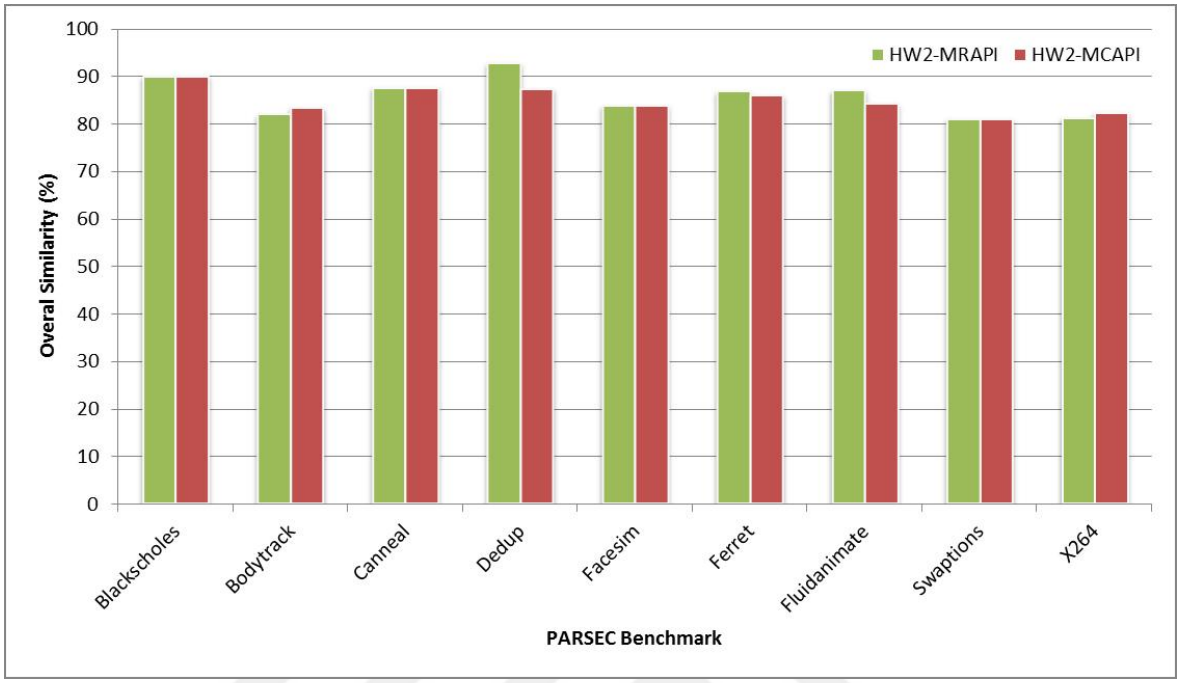

Figure 3.24. Overall similarity scores of synthetic benchmarks from PARSEC for MRAPI/MCAPI on HW2.

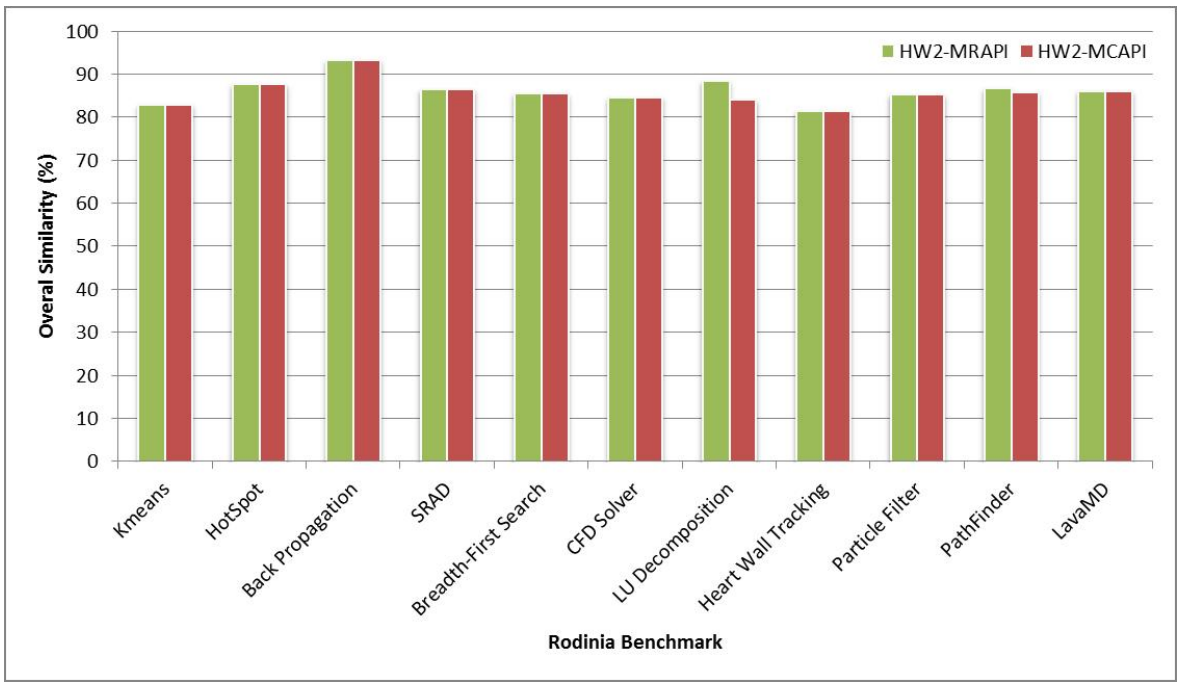

Figure 3.25. Overall similarity scores of synthetic benchmarks from Rodinia for MRAPI/MCAPI on HW2.

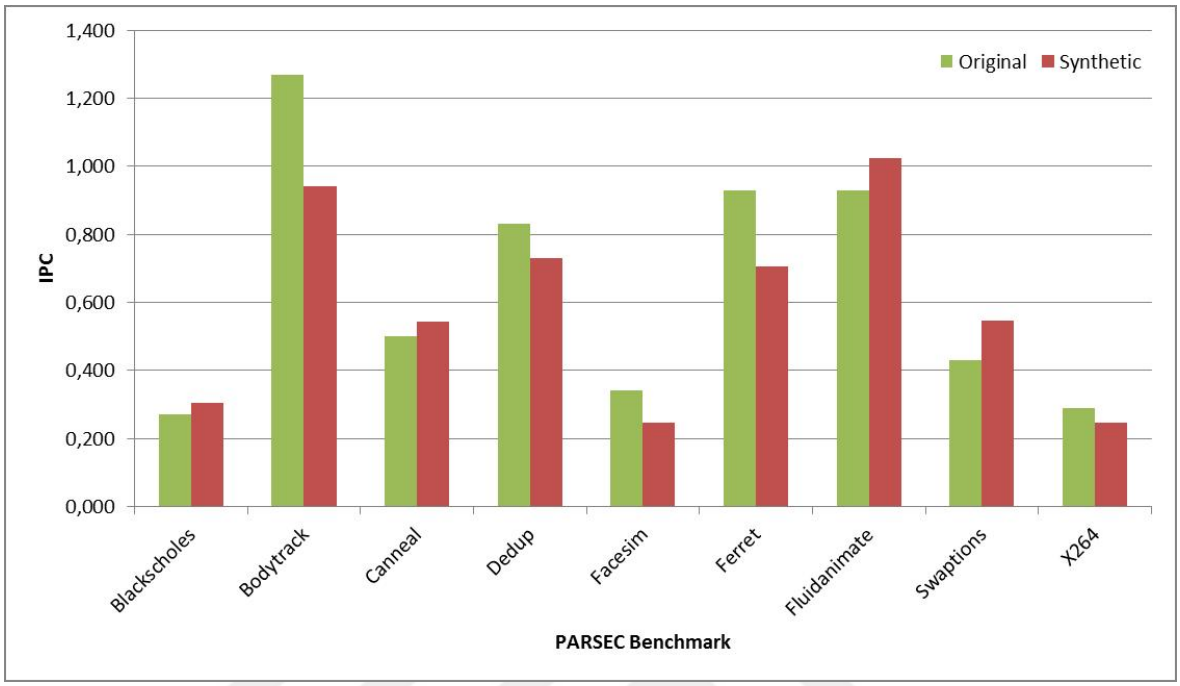

Figure 3.26. Comparison of IPC between the synthetic and the original benchmarks from PARSEC on HW2.

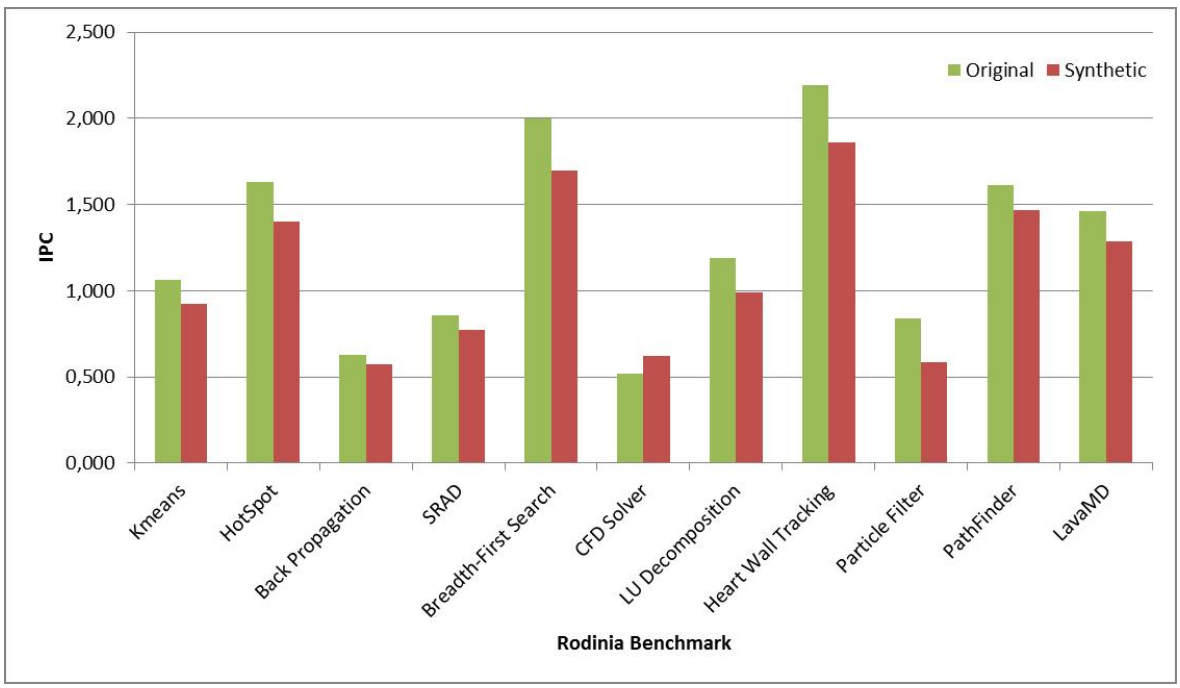

Figure 3.27. Comparison of IPC between the synthetic and the original benchmarks from Rodinia on HW2.

models, hence there is a need for high speed. However, when design matures, original benchmarks should be used for final accurate performance evaluation. When using synthetics for design space exploration, performance evaluation, or bottleneck identification, one should note the characteristics that are kept similar in synthetics with respect to the originals and use synthetics for performance evaluation of such characteristics. For example, our synthetics should not be used for compiler optimization studies since our code blocks do not necessarily perform useful computation hence they can be removed by a compiler. In addition, we do not preserve the code complexity of an original application, hence one should not use a synthetic for code complexity analysis. Since some applications have inherently many inputs some of which could not be covered in the early design stage, generating different synthetics for each input as we did in Section 3.7.4 cannot solve the input change issue. We also assume that the application has a well-defined parallel pattern as synthetic generation is based on parallel patterns. Hence, in order to generate simpler and accurate benchmark, one should follow the parallel patterns completely. Note that all the benchmarks that we used utilizes only a single parallel pattern and we are not aware of a benchmark suite with multiple patterns.

Our experiments showed that synthetic benchmarks are portable across different architectures including different number of processors and cache sizes. We observed that if the number of cores or cache sizes are increased by more than 2× or decreased by more than 0.5× from the base configuration that the synthetic was generated on, then our synthetic may no longer be portable. This is because big changes in architectures result in big changes in CPU stall times and cache conflicts and preserving these changes in the synthetic benchmark requires collecting a large number of characteristics, which is costly and results in slower synthesis process. Whereas, we capture an application's behaviors with just a few characteristics at the cost of potentially reduced sensitivity to architecture changes. For example, the CMR of `Ferret` changes from 33.1 to 14.1 and the CMR of the corresponding synthetic changes from 29.0 to 20.7 going from a system with 6 MB cache to a new system with 20 MB cache ($> 2\times$ change). Although CMRs of both the original and the synthetic decrease, the rate of decrease is not similar. Finally, we note that the portability of a synthetic benchmark also depends on the

application that it is generated from. For example, if an application uses only 1 MB of 4 MB system cache, moving the application to a new system with 20 MB cache does not break the portability of the synthetic benchmark.

3.8. Summary

We developed a fully automated framework, MINIME, capable of generating infrastructure independent synthetic multicore benchmarks. Our main novelty comes from using *parallel patterns* for generating synthetics. These high level characteristics are essential in capturing the behavior of multicore applications such as data sharing and thread communication. Our synthetic benchmarks are readable and portable since they are generated in a high level programming language, C, as opposed to assembly in earlier works. Also, they can use either Pthreads or MCA libraries. Synthetic benchmarks are suitable for embedded multicore systems and can run on any given infrastructure thanks to using MCA libraries.

We experimentally validated MINIME on both x86 and Power Architecture systems using PARSEC, Rodinia, and EEMBC MultiBench benchmark suites. Experiments show that our synthetic benchmarks are similar with the original benchmarks with respect to several metrics. They are also faster (on average 21×) and smaller (on average 14×) than original benchmarks and they mimic the behavior of the original on different microarchitectures. We performed correlation studies to determine the importance of correct parallel patterns in achieving high similarity. We also studied the impact of input changes on the synthetics.

4. THREAD-LEVEL SYNTHETIC BENCHMARKS FOR MULTICORE CPUs

4.1. Overview

The microprocessor industry including mobile, desktop, and server platforms has moved towards multicore architecture design. To take full advantage of multicore CPUs, CPU applications should rely on thread-level parallelism. As a result, multi-threaded applications have been widely used in many domains including scientific and commercial applications. Hence, when we are generating synthetic benchmarks from these multi-threaded applications, we need new techniques for accurate and effective multi-threaded benchmark synthesis.

MINIME tool can generate synthetic benchmarks for multicore systems. However, the accuracy of our synthetic benchmarks generated in Chapter 3 are lower since we do not preserve the characteristics of individual threads in synthetic benchmarks. In this chapter, we use hardware performance counters to keep an aggregated version of characteristics that counts for all threads of the process rather than obtaining counts per-thread. Based on these counter values, certain code blocks are added to the main function of the synthetic benchmark. We call those synthetics as *application-level synthetics*. In this chapter, we preserve the characteristics of individual threads in the synthetic benchmark and implement our solution in the MINIME tool. We call these new synthetics as *thread-level synthetics*. Thread-level synthetics preserve the characteristics of individual threads by using hardware performance counter results for each thread. Then, this accurate information is used to add code blocks per thread, which is more challenging than in application-level synthetics, where an aggregate code block is added to the main function only.

We exploit parallel patterns in thread-level synthetic benchmark generation as in application-level synthetic benchmark generation. We use a new decision tree based

pattern recognition algorithm with lower characterization overhead and faster speed than k-nearest neighbor based approaches used in Chapter 3.

We perform experiments using PARSEC and Rodinia benchmark suites and generate new thread-level synthetics for applications in these suites. Experiments show that our synthetics are more accurate than application-level synthetics, where the average thread similarity score is 84% for thread-level synthetics versus 44% for application-level synthetics. Our synthetic benchmarks are also faster (on average 147×) and smaller (on average 11×) than the original benchmarks that they are generated from.

We first published our results on thread-level synthetic CPU benchmark generation in [24]. In particular, this chapter makes the following contributions.

- We present an algorithm for thread-level synthetic multicore benchmark generation.
- We compare application-level and thread-level synthetic benchmark generation results where thread-level synthetic benchmarks are more similar to the original benchmark that they are generated from.
- We demonstrate that we can generate multi-threaded synthetic benchmarks for real-life PARSEC and Rodinia benchmarks, while being faster (on average 147×) and smaller (on average 11×) than originals.

4.2. Thread-level Synthetic Benchmark Development Framework

Figure 4.1 shows the architecture of our MINIME (thread-level) tool, which we adapted for thread-level synthesis. The tool has three modules, namely, *benchmark characterizer, parallel pattern classifier*, and *benchmark generator*, where the benchmark characterizer captures the important characteristics for each thread of a given original application, the parallel pattern recognizer detects the parallel pattern of the application using the captured characteristics, and the benchmark synthesizer automatically generates a thread-level synthetic benchmark using the detected parallel pattern. In this thread-level synthesis chapter, we use decision trees technique to determine the

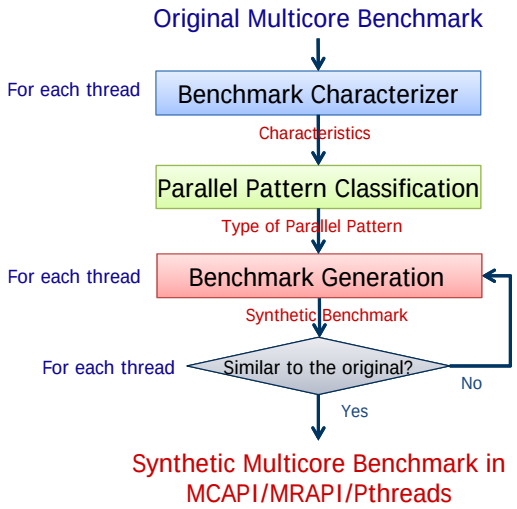

Figure 4.1. MINIME (thread-level): multi-threaded benchmark development framework.

parallel pattern of a given application. We describe this technique in the experiments section. We will now explain benchmark characterizer and benchmark generator components.

4.2.1. Benchmark Characterizer

The benchmark characterizer module collects characteristics of the original application using a combination of dynamic binary instrumentation and performance monitoring counters as we described in Chapter 3. In the previous version of MINIME, we use Linux perf tools [47] to query hardware performance counters at the application level. Application level characterization cannot capture all characteristics of multi-threaded applications because threads share hardware resources such as processor, last level cache and characteristics of each thread impact the performance of multicore applications. Thread-level characterization is crucial for developers and researchers to develop efficient multicore hardware and software. Therefore, there is a need for a new type of thread-level characterization of multi-threaded applications.

In order to add support for thread-level characteristics collection, we add support for PapiEx tool [48] in this chapter. PapiEx is a command line utility to measure per-thread and application level hardware performance counters with Performance Ap-

plication Programming Interface (PAPI) [49]. We collect instructions per cycle (IPC), cache miss rate (CMR), and branch misprediction rate (BMR) characteristics with PapiEx per thread. These are the most commonly used performance counters in the literature.

We use the same characteristics described in Chapter 3 except thread communication. Now, we describe thread communication characteristics where we use *Ratio of Communicating Threads (RCT)* and *Ratio of Communication Volume (RCV)* sub-characteristics. These sub-characteristics are derived using the total number of threads, the number of communicating threads, the total number of cachelines used, and the number of cachelines used in communication. We say that two threads communicate if one thread reads a cacheline written by the other thread.

Definition 4.1 (ratio of communicating threads)**.** *Ratio of Communicating Threads (RCT) is* (the number of communicating threads / the total number of threads).

Definition 4.2 (ratio of communication volume)**.** *Ratio of Communication Volume (RCV) is* (the number of cachelines used in communication / the total number of cachelines used)

4.2.2. Benchmark Generator

The benchmark generator module generates a multi-threaded synthetic benchmark using the parallel pattern type of the application and the performance counter values as described above. However, thread-level synthesis process differs from application-level process as follows. Benchmark generator works iteratively and improves the similarity between the behaviors of the original and synthetic threads at each iteration until a given threshold is reached. During each iteration, code blocks are added for each thread for the performance counter values that are not similar starting from the most dissimilar ones. At the end of the iterations, each thread in the synthetic benchmark preserves performance characteristics of the corresponding thread in the original multi-threaded benchmark leading to accurate synthetics.

```
1 /* code block to increment IPC */
2 for (i = 0; i < WORK_SIZE; i++) {
3   ires = i1 + i2; /* int i1, i2; */
4 }
5
6 /* code block to decrement IPC */
7 for (i = 0; i < WORK_SIZE; i++) {
8   tdres = d1 / d2; /* double d1, d2; */
9 }
```

Figure 4.2. Code block to increment/decrement IPC.

We now define formally our similarity metrics for each of the IPC, CMR, and BMR characteristics. We calculate iss_IPC, iss_CMR, and iss_BMR by using individual similarity score formula. We define *thread similarity score* (tss_i) for each thread i in a benchmark as follows.

Definition 4.3 (thread similarity score)**.** *tss_i = (iss_IPC_i + iss_CMR_i + iss_BMR_i) / 3, where iss_IPC_i is the IPC similarity of thread i, iss_CMR_i is the CMR similarity of thread i, and iss_BMR_i is the BMR similarity of thread i.*

Similarly, we define *average thread similarity score* ($atss$) as follows.

Definition 4.4 (average thread similarity score)**.** *Given n threads, $atss = (tss_1 + \ldots + tss_n)/n$.*

In Chapter 3, we define the overall similarity score, which is an application-level similarity score, as $oss_{app} = (iss_IPC + iss_CMR + iss_BMR)$ / 3. As we will show in the next section, having a high oss_{app} value does not mean having high tss values. Whereas, having a high tss value not only implies having a high oss_{app} value but also a more accurate synthetic.

At each iteration, we check whether every thread's similarity score meets a user defined tss. If not, then for each thread we find the metric with the minimum score. For example, if the minimum score is for IPC, then we insert a code block as shown

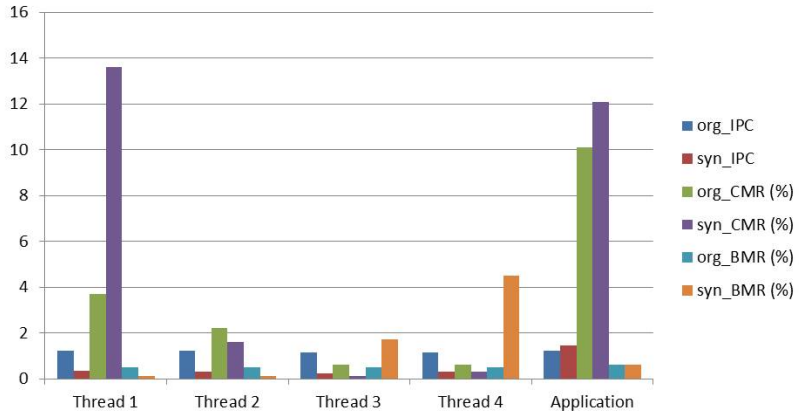

Figure 4.3. Characteristics of `Blackscholes` benchmark and its application-level synthetic.

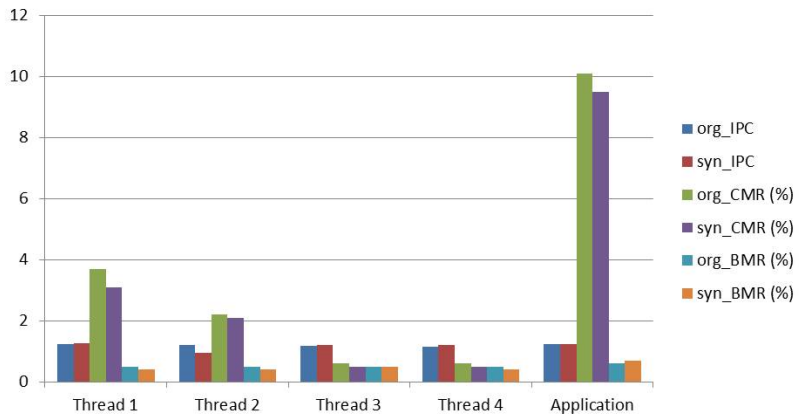

Figure 4.4. Characteristics of `Blackscholes` benchmark and its thread-level synthetic.

in Figure 4.2 either to increment or decrement IPC of synthetic benchmark. Similarly, we add code blocks for CMR and BMR metrics.

4.3. Application-level versus Thread-level Synthetic Benchmarks

In this section, we show the advantage of using thread-level synthetic benchmarks over application-level benchmarks using the `Blackscholes` benchmark from PARSEC benchmark suite. The application has 4 threads.

In Figure 4.3, we show the IPC, CMR, BMR performance characteristics of the original `Blackscholes` benchmark (denoted by org_IPC, org_CMR, org_BMR) and

the application-level synthetic of `Blackscholes` (denoted by syn_IPC, syn_CMR, syn_BMR), respectively.

In application-level synthetic, aggregate values of performance counters are collected during characterization and these values are preserved in the synthetic benchmark by adding code block to the main function. That is, although application-level performance characteristics are preserved (as seen in the plot denoted by Application), performances characteristics of each thread is not preserved and show huge differences. For example, while CMR of Thread 1 of the original benchmark org_CMR is 3.7, CMR of Thread 1 of the synthetic benchmark syn_CMR is 13.6. This demonstrates that application-level similarity generates inaccurate synthetics and does not preserve thread-level behaviors

Figure 4.4 shows the performance characteristics of the original `Blackscholes` benchmark and thread-level synthetic of `Blackscholes`. During thread-level synthesis, values of performance counters are collected for each thread individually and these values are preserved in the synthetic benchmark by adding code blocks to the function of each thread. We now see that thread-level similarity not only provides accurate synthetics by preserving the behavior of each original thread in the synthetic but it also preserves the overall behavior of the original application (application-level behavior) as well.

The advantages of using our new thread-level synthetic benchmark generation technique over our application-level synthetic benchmark generation technique are as follows. Our new technique generates more similar (accurate) synthetic benchmarks in terms of thread-level and application-level. Also, our new technique generates faster synthetic benchmarks compared to our technique described in Chapter 3 as will be shown in the experiments. On the other hand, using our new technique over our technique described in Chapter 3 has some potential downsides. Since we add code blocks to each thread instead of adding them only to the main thread as we do in Chapter 3, our thread-level synthetic benchmarks can be larger than our application-level synthetic benchmarks in terms of lines of code. Similarly, generating thread-level

Table 4.1. Multicore machine configuration.

Parameter	System
#Cores	4
DRAM	6 GB
L1 I/D	32 KB, 8 way
L2	256 KB, 8 way
LLC (L3)	6 MB, 12 way
Branch Predictor	2-level correl, 64-bit
Architecture	x86, Core i7 64-bit

synthetic benchmarks requires more iterations because collecting and preserving per thread characteristic is more costly.

4.4. Experiments

We performed experiments to validate our thread-level synthetic generation technique implemented in MINIME (thread-level) tool. The tool and the benchmark results can be downloaded from our website[2] .

Table 3.6 shows the details of the multicore machine configuration where we ran our experiments. Our multicore machine uses an Intel i7 processor with 4 cores and 6MB cache. Intel SpeedStep® and Hyper-Threading technologies were enabled and threads were not pinned to cores during our experiments. As the compiler tool set, we use GCC 4.6.1 for x86_64 Ubuntu Linux. We compile original benchmarks with default options, and we use '-O0' option for synthetic benchmarks. This is because we generate code blocks for synthetic benchmarks in C and enabling optimizations can result in removing these code blocks.

We used PARSEC [3] and Rodinia (OpenMP) [4] benchmarks as original benchmarks and generated synthetic benchmarks that use POSIX Pthreads, MRAPI, or

[2]http://depend.cmpe.boun.edu.tr/tools/minime

Table 4.2. Benchmark characteristics and pattern classification results.

Original Benchmark					
Suite	**Benchmark**	**LOC**	**#wThreads**	**Known**	**Classified**
PARSEC	Blackscholes	1262	8	TP	TP
	Bodytrack	7696	9	GD	GD
	Canneal	2794	4	TP	GD
	Dedup	7125	8	Pl	Pl
	Facesim	20275	5	TP	TP
	Ferret	10765	18	Pl	Pl
	Fluidanimate	2784	4	GD	GD
	Swaptions	1095	4	TP	TP
	X264	38546	15	Pl	Pl
Rodinia	Kmeans	2146	3	TP	TP
	HotSpot	196	3	GD	GD
	Back Propagation	478	7	TP	TP
	SRAD	495	1	TP	TP
	Breadth-First Search	125	3	TP	TP
	CFD Solver	1539	7	TP	GD
	LU Decomposition	541	3	GD	GD
	Heart Wall Tracking	2244	3	TP	TP
	Particle Filter	398	7	GD	GD
	PathFinder	127	3	GD	GD
	LavaMD	353	3	GD	GD

MCAPI parallel libraries. We used the simmedium input for PARSEC and default input for Rodinia. For each benchmark, we ran the original and the synthetic benchmarks 10 times in order to obtain characteristics. We also set tss_i to 80% for each thread and iteration threshold to 100.

Table 4.2 shows benchmark characteristics and our pattern classification results. The column *Known* shows the pattern of the benchmark known from the literature

Table 4.3. Thread-level synthetic benchmark generation results.

Original Benchmark		Application-level				Thread-level			
Suite	**Benchmark**	**#iter**	**LOC**	**Speedup(×)**	**atss**	**#iter**	**LOC**	**Speedup(×)**	**atss**
PARSEC	Blackscholes	2	124	10	24	50	431	21	89
	Bodytrack	16	403	11	28	74	728	127	82
	Canneal	2	136	22	76	25	409	78	82
	Dedup	9	440	36	61	60	613	11	85
	Facesim	5	127	15	30	98	250	475	83
	Ferret	5	1426	67	18	81	1222	344	80
	Fluidanimate	2	330	15	43	16	368	18	82
	Swaptions	2	144	13	66	100	306	146	84
	X264	12	940	26	60	80	553	20	80
Rodinia	Kmeans	15	180	36	39	36	245	778	85
	HotSpot	10	195	16	57	49	220	22	81
	Back Propagation	5	129	20	56	42	547	15	83
	SRAD	17	222	12	59	32	104	66	81
	Breadth-First Search	18	195	35	23	85	263	180	87
	CFD Solver	11	243	10174	37	93	352	2593	91
	LU Decomposition	32	199	10	38	94	138	21	81
	Heart Wall Tracking	16	180	34	33	45	286	114	89
	Particle Filter	9	242	10	30	80	476	125	84
	PathFinder	14	343	13	61	88	551	27	85
	LavaMD	19	274	30	37	23	230	206	83

and column *Classified* shows the pattern of the benchmark we classified. We show the number of the worker threads (*#wThreads*) in original benchmarks which will be kept the same in synthetic benchmarks. The lines of code is denoted by (*LOC*).

Table 4.3 shows our thread-level synthetic benchmarks that use POSIX Pthreads API. We also generated synthetic benchmarks that use MCAPI and MRAPI APIs and obtained similar results. We also show results for application-level benchmarks from Chapter 3 for comparison purposes. In the table, we show the number of iterations (*#iter*) it takes to generate the synthetic benchmark with the required target similarity percentage. The column *Speedup(×)* shows the speedup in terms of running time and the column *atss* shows the average thread similarity score for the given benchmark.

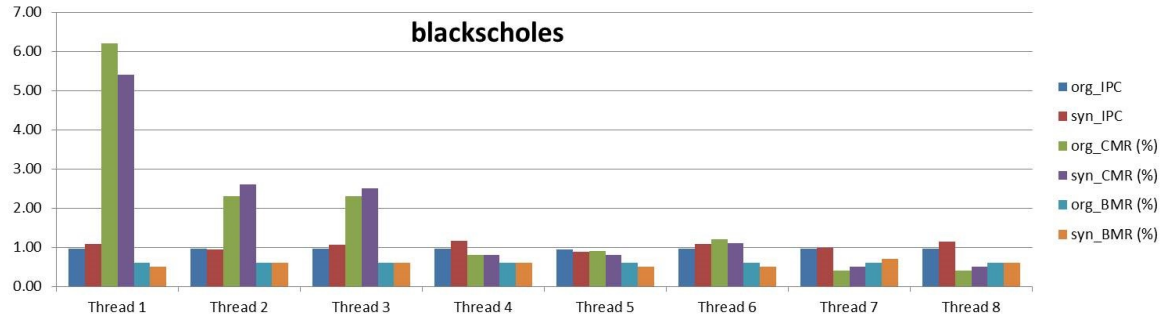

Figure 4.5. Characteristics of `Blackscholes` benchmark and thread-level synthetic of Blackscholes with 8 threads.

On average, we generate the synthetic benchmark that satisfies the target score in 63 iterations. It takes more iterations when performance characteristics of a thread is too low or too high such as the BMR of `Swaptions` is 10% which leads to high influence between threads. We have maximum 2593× speedup in `CFD Solver` since the running time of original benchmark is longer (337.15 seconds) than the others. The average speedup is 147× without `CFD Solver` and it is 269× with `CFD Solver`. This demonstrates that we have large speedup values when the running time of original application is high. Similarly, the code size is reduced in synthetics. We have maximum 81× reduction in lines of code for `Facesim` and on average we have 11× code size reduction. Since there is no code for communication in task parallel synthetic benchmarks, their code sizes are smaller than other synthetics.

In Figure 4.5, we show the microarchitecture dependent characteristics of original `Blackscholes` benchmark and thread-level synthetic of `Blackscholes` with 8 threads. In the figure, for each thread we show IPC, CMR, and BMR of original and synthetic benchmark. *tss*s of threads are 85, 95, 93, 93, 89, 88, 85, and 86, respectively. The average thread similarity score (*atss*) of `Blackscholes` benchmark is 89. Our technique is applicable for arbitrary number of the threads for example the results for `Blackscholes` with 4 threads was shown in Figure 4.4. Next, we show the detailed similarity scores for all benchmarks.

In Figure 4.6, we give average *tss*s for all benchmarks and we show the minimum and maximum *tss*s as error bars. The maximum *atss* is 91 in `CFD Solver` and the av-

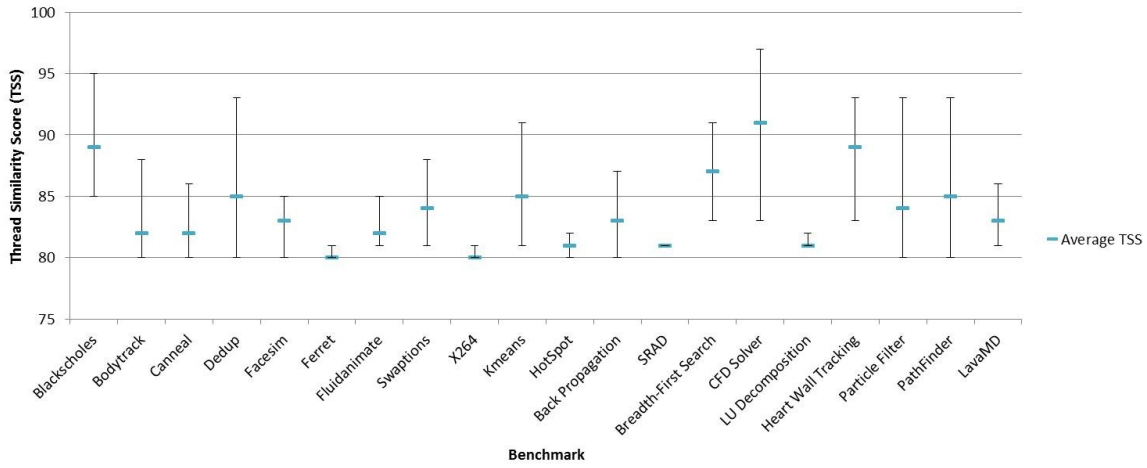

Figure 4.6. Average, maximum, and minimum thread similarity scores of all thread-level synthetic benchmarks.

erage of all *atss*s is 84. The minimum *atss* is 80 in `Ferret` and `X264`. This is because the original benchmarks have very high BMR (7.6%) values compared to other benchmarks and converging benchmarks with very high or low values of microarchitecture dependent characteristics is harder than others. The figure shows that threads in the synthetic and the original benchmarks are similar to each other over 80%, which was set by the user. Currently, 80% is the threshold we set for *atss* since a higher value is not possible with the code blocks that we generate. This can be improved if we employ low level code blocks such as assembly but since this will prevent our code from being portable we decided not to pursue this route. In any case, the obtained results show the high quality of thread-level synthetics generated by our automated framework

We now compare our new thread-level synthetic benchmarks with our previous application-level synthetic benchmarks. We use the average thread similarity score of both types of synthetic benchmarks for comparison. We ran our new characterizer on the application-level synthetics that are generated in Chapter 3 in order to collect their thread-level characteristics. Figure 4.7 shows our results. It is clear from the figure that thread-level synthetics have much better average thread similarity score (84% on average) than application-level synthetics (44% on average). Hence, the accuracy of our new thread-level synthetic benchmarks is much better than earlier application-level synthetic benchmarks.

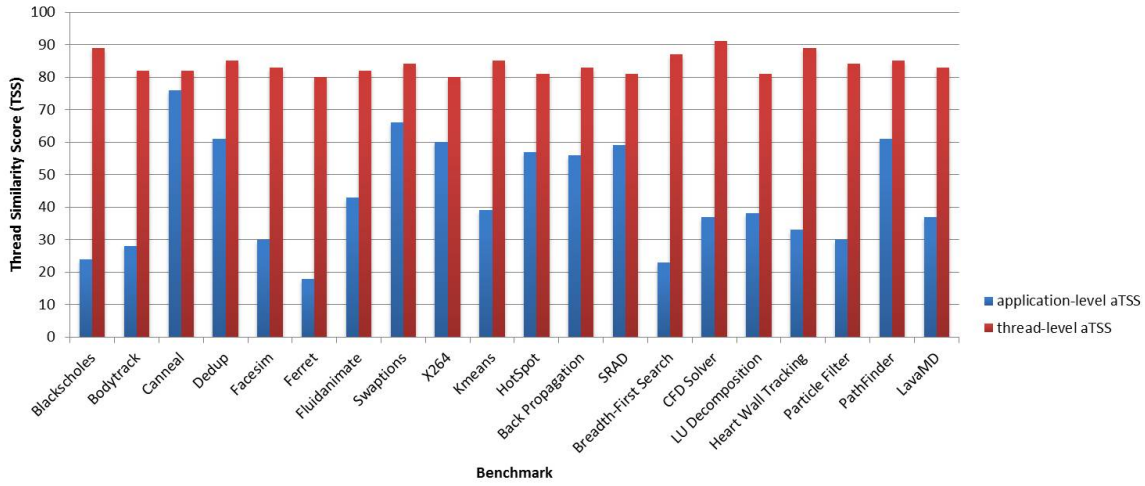

Figure 4.7. Comparison of average thread similarity scores for application-level and thread-level synthetic of all benchmarks.

4.4.1. Decision tree based parallel pattern recognition

In this section, we describe an automated way of recognizing parallel patterns described in Section 4.2. There exist several machine learning techniques for data classification in the literature. Each technique has advantages and disadvantages. For example, classification with decision trees is memory efficient and fast compared to the memory intensive kNN. We use a parallel pattern recognition technique using the k-nearest neighbor (kNN) technique in Chapter 3 where we use 12 characteristics with an accuracy of 100%. Given the advantages of decision trees, we want to improve the performance of our parallel pattern recognition technique. As we will experimentally show, the decision tree technique requires fewer number of characteristics compared to the kNN technique. Hence, characterization overhead of the decision tree technique is lower and it runs faster. We describe the details of using decision trees for pattern recognition in Chapter 6.

We show the parallel pattern classification results of our decision tree in column *Known* of Table 4.2. Our decision tree correctly classifies 18 of the 20 benchmarks in the test set with a 90% accuracy. Hence, the accuracy of our decision tree is high. We define the characterization overhead as the ratio of the running time of a multi-threaded application to the un-instrumented running time of the same application.

Similarly, we define the speed as the amount of time needed for an algorithm to perform learning and classification. The characterization overhead is 3.5×, since we use only four sub-characteristics for classification instead of twelve and the speed is 0.01 seconds. Whereas, for our pattern recognition results with kNN technique the characterization overhead is 20.2×, since we use all 12 characteristics and the speed is 0.04 seconds.

4.5. Summary

We describe a new thread-level synthetic benchmark generation framework that generates synthetic benchmarks from the benchmarks given in an existing benchmark suite. Our synthetics not only preserve the performance behaviors of individual threads in existing benchmarks unlike earlier works but they are much faster (average 147× speedup) and smaller (average 11× reduction) than originals. Our thread-level synthetics have also better accuracy than the earlier application-level synthetics. We also developed a new decision tree based parallel pattern recognition technique that is faster than kNN parallel pattern recognition technique.

5. SYNTHETIC BENCHMARK GENERATION FOR GPUs

5.1. Overview

GPUs have become increasingly a popular platform for data parallel applications thanks to their high parallel throughput and high memory bandwidth. GPUs present high performance but they require optimizations to achieve this high performance. In addition, GPU applications demonstrate different characteristics from CPU applications since GPUs have significantly different architectures from CPUs. Hence, in early design exploration of GPUs, it is important to have GPU specific benchmarks that have similar performance characteristics with the applications that will run on the GPU.

We present a novel synthetic benchmark generation approach for GPUs. Our approach is capable of generating synthetic benchmarks that are small, fast, and they accurately mimic the characteristics of the original applications they are generated from. They can be used for early performance studies of GPUs in both actual hardware and simulation. As described in [50], CPU simulation acceleration techniques such as sampling and statistical simulation cannot be easily applied to GPUs since each thread in a GPU application executes a small number of instructions compared to CPU applications. Our approach helps developers and researchers focus on analyzing the synthetic benchmark results by hiding the difficulties of benchmark development from them.

Our fully automated synthetic benchmark generation approach comprises of two steps: (1) characterizing a GPU application to capture its inherent characteristics and modeling the captured application characteristics into an abstract benchmark model, (2) generating a synthetic GPU benchmark using the abstract benchmark model. We generate our synthetic benchmarks using OpenCL [32], which is a framework for developing applications that run across heterogeneous platforms consisting of CPUs, GPUs,

and DSPs.

During benchmark characterization, unique behaviors of an application of interest are captured as a set of quantifiable abstract characteristics. Accurate synthetic benchmark generation requires a sufficient number of characteristics to capture and model the behaviors of an existing application. The characteristics that we use to capture the behavior of a GPU application are instruction throughput, compute unit occupancy, computation-to-memory access ratio, memory instruction mix, and memory efficiency. In other words, we speed up GPU architectural simulation by generating synthetic benchmarks from existing benchmarks that mimic these characteristics. These characteristics are widely used in the literature [15–19]. We also apply principal component analysis to find the most important characteristics. Once we characterize and model the original application as an abstract benchmark model, we generate a synthetic benchmark from this model. The synthetic benchmark consists of host (CPU) and compute device (GPU) code and it is generated in C++ using OpenCL library. Our synthetic benchmarks preserve all of the characteristics captured from the original application and they can be executed either on a simulator or a target platform.

Our fully automated benchmark synthesis framework for GPUs is called MINIME-GPU and we experimentally validate the efficiency of our approach using our framework. MINIME-GPU is an extension of our tool MINIME targeting synthetic CPU benchmark generation. During the experiments, we use Multi2Sim [1] simulator to collect the characteristics of GPU applications. We generated synthetic benchmarks from AMD benchmark suite [51] for AMD Southern Islands GPUs [52] that are available with Multi2Sim distribution [1]. The experimental results show that our synthetic benchmarks mimic the characteristics of the original applications they are derived from where the average similarity is 96% and average speedup is 541×. We also experimentally validate that our synthetic benchmarks mimic the behaviors of the originals across different architectures on both the Multi2Sim simulator as well as on real GPU hardware. Furthermore, they are human readable and cannot be reverse engineered to create the original code or algorithms.

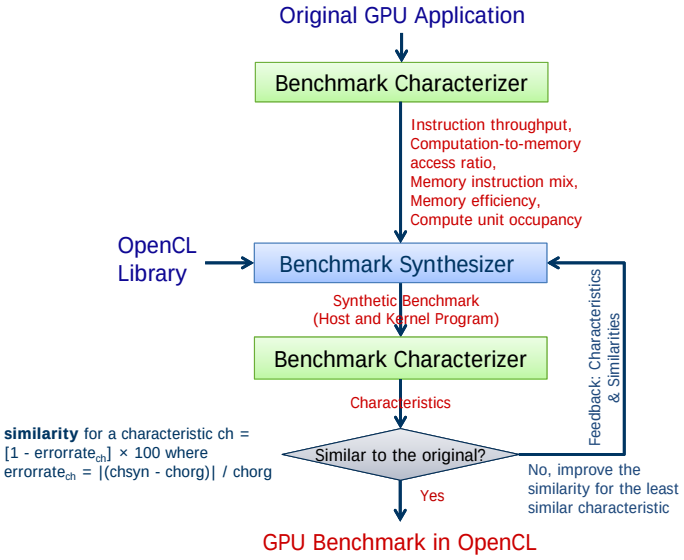

Figure 5.1. MINIME-GPU: multicore benchmark synthesizer for GPUs.

We first published our results on synthetic GPU benchmark generation in [25]. In summary, this chapter makes the following contributions.

- We use principal component data analysis methodology to identify critical GPU application characteristics.
- A synthetic benchmark generation framework is proposed and implemented to generate synthetic OpenCL benchmarks for GPUs from a given GPU application.
- Our synthetic GPU benchmarks are portable, human readable, smaller, and faster than the original applications that they are generated from.
- Our synthetic GPU benchmarks do not compromise the proprietary nature of the original applications since one cannot obtain the original application from our synthetic benchmarks by reverse engineering.
- The experimental results showed that our synthetic benchmarks mimic the characteristics of the original applications they are generated from across different architectures where the average similarity is 96% and average speedup is 541×.

5.2. High-level Framework

Figure 5.1 shows our fully automated high-level benchmark synthesis framework for GPUs, called MINIME-GPU. Our framework contains two main modules: *bench-*

mark characterizer and *benchmark synthesizer*. Benchmark characterizer captures the characteristics of a GPU application and generates an abstract GPU benchmark model from these characteristics. Benchmark synthesizer, firstly, generates a candidate synthetic GPU benchmark from generated abstract benchmark model. Then, benchmark synthesizer iteratively calculates the similarity between the original application and synthetic benchmark and improves the similarity by generating new synthetic benchmarks. This approach is similar with the MINIME framework proposed in Chapter 3 for multicore CPU benchmark synthesis. However, this approach differs from the MINIME by targeting GPU architectures, using GPU specific characteristics and generating synthetic benchmarks using OpenCL library. We further discuss this in the related work section. Note that in this chapter, we generate synthetic benchmarks that only preserve the characteristics of kernel programs, that is, we do not mimic the characteristics of host programs as in [50]. This is because we target the GPU applications in which the whole solution of the problem is implemented on the compute device side. Also, we are not aware of a benchmark suite in which some parts of a problem are solved on the host side and the other parts are solved on the compute device side. In any case, for such CPU/GPU benchmarks, we can use MINIME (CPU) to generate synthetic benchmarks for host programs and MINIME-GPU to generate synthetic GPU benchmarks for kernel programs.

5.3. Benchmark Characterization

When generating a synthetic benchmark, the efficacy of benchmark characterization to capture the behaviors of an original application is crucial. This is because only the behaviors captured from the original application can be preserved in the corresponding synthetic benchmark. Hence, we design a benchmark model to cover the most crucial characteristics that capture the major behaviors of a GPU application. In our model, we use instruction throughput, computation-to-memory access ratio, memory instruction mix, memory efficiency, and compute unit occupancy characteristics. Note that we also use these characteristics to determine program similarity as we show in Section 5.4. These characteristics are widely used in the literature and they effectively

Table 5.1. GPU benchmark characteristics.

Group	Characteristics
Instruction throughput	Instruction per cycle (IPC)
Computation-to-memory access	Computation-to-memory access ratio (CMAR)
Dynamic memory instruction mix	Private memory ratio
	Local memory ratio
	Global memory ratio
Memory efficiency	Memory coalescing
	Hit ratio
Compute unit occupancy	Work-items per work-group
	Registers per work-item
	Local memory per work-group
	Number of (in-flight) wavefronts

capture the behaviors of GPU applications [15–19]. Also, we validate the effectiveness of these characteristics using Principal Component Analysis (PCA) as shown in the experiments section.

Now, we describe each characteristic shown in Table 5.1 in detail.

- *Instruction throughput* represents the total throughput of an application and we use **instruction per cycle (IPC)** to measure it.
- *Computation-to-memory access ratio (CMAR)* is the ratio of computations (the number of scalar, vector, and branch instructions) to memory accesses (the number of private, local, and global memory operations). We use this characteristic to determine if an application is compute-insensitive or memory-insensitive, where a higher CMAR indicates a compute-insensitive application and a lower CMAR indicates a memory-insensitive application.
- *Dynamic memory instruction mix* is the distribution of memory instruction types that are executed. We use **private**, **local**, and **global memory ratios** to de-

termine the memory instruction mix. We measure private memory ratio as the number of private memory instructions executed divide by the total number of memory instructions executed. Similarly, we calculate local and global memory ratios. These characteristics are crucial for performance because different memory instructions have different throughputs. For example, a higher global memory ratio can result in poor performance and scalability.

- *Memory efficiency* is measured by using memory coalescing and hit ratio. **Memory coalescing** refers to combining multiple memory accesses into a single combined access. Since fewer requests result in less contention to global memory, a high ratio of coalesced memory accesses improves application performance. Hence, memory coalescing can be the first optimization to consider in which memory bandwidth usage is reduced. Note that the maximum memory coalescing can be 1 when all accesses are coalesced and the minimum memory coalescing can be 0 when there is no coalesced access. Also, we measure **hit ratio** for caches, TLBs, and main memory, which is the number of hits divided by the number of accesses. Similarly, a higher hit ratio results in a high performance and the hit ratio ranges from 0 to 1.
- *Compute unit occupancy* refers to the utilization of the computation resources (wavefronts) of a compute unit on a GPU. **Work-items per work-group (work-group size)**, **registers per work-item**, and **local memory per work-group** limit the **number of (in-flight) wavefronts** per compute unit. Note that a higher compute unit occupancy indicates a higher utilization of computation resources. In order to mimic the compute unit occupancy behavior of an original application, we need to capture these characteristics.

During the characterization of a GPU application, we analyze the final executable binary files instead of analyzing the source code. Hence, we do not need the source code of an original application, hence our approach can work on proprietary applications.

```
1  Benchmark_syn() {
2    /* Constructor: Initialize class member variables. */
3  }
4
5  int setup() {
6    /* Perform all benchmark setup */
7    setupBenchmark_syn(); /* Allocate and initialize host memory. */
8    setupCL(); /* Perform OpenCL related initializations:
9    * create kernel0 from syntheticKernel0, create context, command queue, etc. */
10 }
11
12 int runCLKernels0() {
13   /* Set values for kernel's arguments, enqueue calls to the kernel,
14    * and wait until the kernel execution is completed. */
15   setWorkGroupSize0(); /* Set the work-group size */
16 }
17
18 int run() {
19   /* Run OpenCL kernel program(s). */
20    runCLKernels0(); /* run kernel0 created in setupCL function. */
21 }
22
23 int cleanup() {
24   /* Cleanup OpenCL API (context, memory buffer, etc.) resources and
25    * program (input/output memory, etc.) resources.  */
26 }
27
28 int main(int argc, char * argv[]) {
29   Benchmark_syn clBenchmark_syn;  // Create a synthetic benchmark object
30   clBenchmark_syn.setup;          // Setup
31   clBenchmark_syn.run();          // Run
32   clBenchmark_syn.cleanup();      // Cleanup
33 }
```

Figure 5.2. Host program of a synthetic benchmark.

```
1  /* matrixA and matrixB are inputs and matrixC is output. */
2  __kernel void syntheticKernel0(__global float4 *matrixA, __global float4 *matrixB,
3        __global float4 *matrixC, uint widthA, uint widthB, __local float4 *blockA)
4  {
5    /* Code blocks to improve the similarities of characteristics.
6     * For example, code block to decrement IPC or to increment memory coalescing */
7  }
```

Figure 5.3. Kernel program of a synthetic benchmark.

5.4. Benchmark Generation

In this section, we elaborate on how we generate a synthetic benchmark from the captured characteristics of an original application, that is the abstract benchmark model. The generated synthetic benchmark consists of a host program and a kernel program. We show the host program with its basic functions in Figure 5.2 and the kernel program in Figure 5.3. The host program has a main function and other functions to setup and run OpenCL kernel(s). In the *main* function, first, we create a (C++) synthetic benchmark object by using the class constructor and then perform setup, run, and cleanup operations on this object, respectively. In the *setup* function, we adjust the width and the height of inputs/outputs and then allocate and initialize host memory. We then create OpenCL constructs including context, device list, command queue, and memory buffers. Note that, we create the OpenCL program construct by using offline compilation mechanism in which we build the kernel program executable offline and then load the binary at runtime. In the *run* function, we execute all kernel program(s) each corresponding to a kernel in the original application. In the *runCLKernels0* function, we set values for the kernel's arguments including inputs, outputs, and sizes such as height and width, enqueue calls to the kernel by using the command queue, and wait until the kernel execution is completed. In the *setWorkGroupSize0* function, we set the work-group size (work-items per work-group) based on the characteristic of the original application. Lastly, in the *cleanup* function, we remove the allocated/created resources including memory, context, and memory buffer.

The kernel program in Figure 5.3 has two inputs (*matrixA* and *matrixB*) and an output (*matrixC*) matrix where matrices can be 1- or 2-dimensional depending on the input/output dimensions of the original application. Also, if a local memory is used by the original application, we define and use a local memory (*blockA*) in the synthetic application. An important feature of our thesis is the code blocks inserted into the kernel to mimic the characteristics of the original application. Next, we describe how we calculate the similarity between an original application and the synthetic benchmark before we elaborate on code blocks.

5.4.1. Similarity Measurement

From the list of characteristics shown in Table 5.1, we use IPC, CMAR, private, local, and global memory ratio, memory coalescing, and local memory per work-group characteristics to calculate the similarity (accuracy) between a synthetic and the original benchmark. We do not use the number of in-flight wavefronts and work-items per work-group because we make sure that a synthetic benchmark has the same values for the number of kernels, the number of work dimensions, global sizes, local sizes, work-items per work-group, and the number of in-flight wavefronts. We also do not use registers per work-item characteristic since due to a bug in Multi2Sim, we can collect this characteristic for all original benchmarks that are available with Multi2Sim distribution but not for synthetic benchmarks that we create. Finally, although we do not use the hit ratio characteristic in similarity calculation, the synthetic benchmarks preserve this characteristic as we will show in the experiments. This is because characteristics such as global memory ratio and memory coalescing implicitly capture (mimic) this characteristic.

We use the individual similarity score to assess the similarity of a synthetic benchmark and an original application. We calculate the overall similarity score as oss_{gpu} = (iss_{IPC} + iss_{CMAR} + $iss_{privateMemoryRatio}$ + $iss_{localMemoryRatio}$ + $iss_{globalMemoryRatio}$ + $iss_{memoryCoalescing}$ + $iss_{localMemoryPerWorkgroup}$) / 7. Thresholds for individual and overall similarity scores are given by the user.

5.4.2. Code (Block) Generation

After we measure the similarity between the original application and the corresponding synthetic benchmark, if the individual and overall similarity scores meet the individual and overall thresholds defined by the user, the synthesis process is completed. Note that our code blocks do not contain instruction set architecture (ISA) specific assembly instructions, as was done in earlier synthetic benchmark generation works, since they break portability. Also, we compile our synthetic benchmarks with '-O0' option so that the compiler did not remove our code blocks. Otherwise we start

```
1   /* CB1: Code block to increment IPC */
2   int ipcI;
3   for (ipcI = 0; ipcI < 1; ipcI++)
4     matrixC[0] = 1.2;
5
6   /* CB2.1: Code block to decrement IPC used if (candIPC - origIPC) > 9 */
7   int xValue = get_global_id(0);    int yValue = get_global_id(1);
8   float4 ipc1 = matrixA[yValue * widthA + xValue];
9   matrixA[0] = ipc1;
10
11  /* CB2.2: Code block to decrement IPC used otherwise */
12  float ipc1 = 1, ipc2 = 2;
13  for (i = 0; i < LOOP_COUNT; i++) { // LOOP_COUNT is variable
14    matrixA[i * 4 + 1] = ipc1 + ipc2;
15    mem_fence(CLK_GLOBAL_MEM_FENCE);
16  }
17
18  /* CB3: Code block to increment CMAR */
19  float4 cmar1 = matrixA[0];
20  matrixC[0] = cmar1 + cmar1; // the number of cmar1 in sum operation can change
21
22  /* CB4: Code block to decrement CMAR */
23  float4 cmar1, cmar2;
24  int cmari;
25  for (cmari = 0; cmari < LOOP_COUNT; cmari++) { // LOOP_COUNT is variable
26    cmar1 = cmar1 + cmar2;
27  }
28  matrixC[0] = cmar1;
```

Figure 5.4. Sample code blocks to increment/decrement the values of kernel program (instruction throughput and computation-to-memory access) characteristics.

```
1   /* CB5.1: Code block to increment private memory ratio
2    * used if #instructions of original application <= 500 */
3   float4 pmr1, pmr2;    matrixC[0] = pmr1 + pmr2;
4
5   /* CB5.2: Code block to increment private memory ratio used otherwise */
6   float4 pmr1 = matrixA[0];    float4 pmr2 = matrixB[0];
7   matrixC[0] = pmr1 + pmr2;
8
9   /* CB6: Code block to increment global memory ratio */
10  for (i = 0; i < LOOP_COUNT; i++) { // LOOP_COUNT is variable
11    matrixC[i] = matrixA[i] + matrixB[i];
12  }
13
14  /* CB7: Code block to decrement global memory ratio */
15  float4 gmr1 = matrixA[0];
16  float4 gmr2 = gmr1 + 2;    float4 gmr3 = gmr2 + 2;
17  float4 gmr4 = gmr3 + 2;    matrixC[1] = gmr4;
18
19  /* CB8.1: Code block to increment memory coalescing used if origMcol > 0.7 */
20  matrixC[0] = blockA[0]; // blockA is allocated from local memory
21
22  /* CB8.2: Code block to increment memory coalescing used otherwise */
23  size_t tid = STRIDE * get_global_id(0); // STRIDE is variable.
24  // if origMcol > 0.4, STRIDE is 2; else if origMcol > 0.1, STRIDE is 3;
25  // otherwise, STRIDE is 4.
26  if (tid < 32) {
27    blockA[tid] = 1.2;
28    matrixC[0] = blockA[tid];
29  }
30
31  /* CB9: Code block to increment or decrement local memory per work-group */
32  __local float localds[LOCAL_MEM_SIZE]; // LOCAL_MEM_SIZE is variable
```

Figure 5.5. Sample code blocks to increment/decrement the values of kernel program (dynamic memory instruction mix, memory efficiency, and compute unit occupancy) characteristics.

iterations, where at each iteration, we add, remove, or change a code block for the characteristic that has the least similarity with the original application in the kernel code. Figures 5.4 and 5.5 show sample code blocks, which we experimentally obtained, to increment/decrement the values of characteristics. Now we explain each of these code blocks.

Code blocks to mimic instruction throughput: For example, when the IPC of an original application is higher than the IPC of the synthetic benchmark, we add CB1, which has instructions with high IPC. We have two different code blocks (CB2.1 and CB2.2) to decrement IPC where we use CB2.1, if the IPC of the synthetic benchmark minus the IPC of the original application is greater than 9. Otherwise, we use CB2.2. Note that both CB2.1 and C2.2 have instructions with low IPC.

Code blocks to mimic computation-to-memory access: We use CB3 to increment CMAR where there exist many computation instructions and a few memory access instructions. In the code block, the number of *cmar1* used in sum operation (*matrixC[0] = cmar1 + ... + cmar1*) can change depending on the CMAR of the original application. For example, if CMAR of an original application is around 0.80, the number of *cmar1* used in sum operation is 1, that is *matrixC[0] = cmar1*, and if CMAR of an original application is around 0.44, the number of *cmar1* used in sum operation is 2, that is *matrixC[0] = cmar1 + cmar1*. Note that we experimentally characterized code blocks and fond the specific values such as 0.80 and 0.44. Although we experimentally validate these values on AMD GPUs, these characteristics are platform independent as will be shown in the experiments. Similarly, we use CB4 to decrement CMAR where there exist many memory access instructions and a few computation instructions. In the code block, the number of iterations ($LOOP_COUNT$) is a variable and depends on the CMAR of the original application. In order to find $LOOP_COUNT$, we initialize it as 1 and then increment it until we meet the CMAR similarity between the original and synthetic.

Code blocks to mimic dynamic memory instruction mix: We use code blocks CB5.1 and CB5.2 in which we perform operations on private memories to in-

crement private memory ratio. If the number of instructions of an original application is less than or equal to 500 instructions, we use CB5.1, otherwise, we use CB5.2 that has more instructions than CB5.1. Hence, using CB5.1 provides higher speedups for small applications since it has less number of instructions and using CB5.2 provides a higher private memory ratio since it has more private memory instructions. Note that the value 500 for the number of instructions is experimentally obtained, that is, when we use CB5.2 in the synthetics for original applications with less than 500 instructions instead of CB5.1, we observe that the synthetics can be slower than the corresponding originals, which is something that we do not want. We use CB6 in which there exist memory operations on global memories such as *matrixA*, *matrixB*, and *matrixC* to increment global memory ratio. In the code block, the number of iterations (*LOOP_COUNT*) can vary depending on the global memory ratio of the original application. In CB7, we perform memory operations on private memories to decrement global memory ratio of the synthetic benchmark. In this code block, we define a number of private memories (*gmr1, gmr2, ..., gmrN*) where the number is chosen high if the difference between the global memory ratios of the original and synthetic is high, otherwise the number is chosen low.

Code blocks to mimic memory efficiency: We have two different code blocks, CB8.1 and CB8.2, to increment memory coalescing of a synthetic benchmark. If the memory coalescing of an original benchmark is greater than 0.7, we use CB8.1; otherwise, we use CB8.2. In CB8.2, *STRIDE* is variable and if the memory coalescing of an original benchmark is greater than 0.4, *STRIDE* is 2; else if it is greater than 0.1, *STRIDE* is 3; otherwise, *STRIDE* is 4. In CB8.1, each work-item accesses the same memory location (*blockA*), hence we have high memory coalescing. On the other hand, in CB8.2, each work-item accesses strided memory locations where stride lengths are 2, 3, and 4, hence they have lower memory coalescing.

Code blocks to mimic compute unit occupancy: We use CB9 to increment or decrement local memory per work-group characteristic of a synthetic benchmark. In this code block, we define a local memory (*localds*) that has the same size (*LOCAL_MEM_SIZE*) with the local memory used in the original application.

```
1   __kernel void syntheticKernel_local0(__global float4 *matrixA,
2                 __global float4 *matrixB, __global float4 *matrixC,
3                 uint widthA, uint widthB, __local float4 *blockA)
4   {
5     /* CB3: Code block to decrement CMAR */
6     float4 cmar1, cmar2;
7     int cmari;
8     for (cmari = 0; cmari < 8; cmari++) {
9       cmar1 = cmar1 + cmar2;
10    }
11    matrixC[0] = cmar1;
12
13    /* CB5.2: Code block to increment private memory ratio */
14    float4 pmr1 = matrixA[0];
15    float4 pmr2 = matrixB[0];
16    matrixC[0] = pmr1 + pmr2;
17
18    /* CB8.1: Code block to increment memory coalescing */
19    matrixC[0] = blockA[0];
20  }
```

Figure 5.6. Synthetic benchmark (kernel program) for `QuasiRandomSequence`.

We continue until the iteration upper bound set by the user is reached or the individual and overall similarity scores are satisfied.

5.4.3. A Detailed Example of GPU Benchmark Synthesis

We demonstrate our approach on a GPU benchmark `QuasiRandomSequence` from AMD APP SDK where we set the overall similarity rate threshold as 90% and the individual similarity score threshold as 60%. Figure 5.6 shows the final synthetic benchmark where it takes 11 iterations to obtain 98% overall similarity score. The individual similarity scores except CMAR are 100% and the CMAR similarity score is 88%. The speedup defined as the ratio of the GPU simulation time of the original application to the GPU simulation time of the corresponding synthetic benchmark is 6.6×. For the initial synthetic benchmark, the overall similarity score is 57% and each one of CMAR, private memory ratio, and memory coalescing similarity scores are the lowest (0%). Hence, during iterations we add (CB3, CB5.2, and CB8.1) code blocks shown

in the figure to obtain the final synthetic benchmark. Specifically, for CB3, we set *LOOP_COUNT* as 1 initially and then update it until we mimic the CMAR of the original application, where it becomes 8. We use CB5.2 to increment private memory ratio because the number of instructions of `QuasiRandomSequence`, which is 32288, is more than 500. Since the memory coalescing of `QuasiRandomSequence`, which is 0.98, is larger than 0.7, we use CB8.1 to mimic the memory coalescing characteristic of the original application. We observe that adding more code blocks decreases speedup as expected, that is, the initial synthetic benchmark is faster than the final one. This indicates that decreasing similarity thresholds can result in faster synthetic benchmarks.

5.5. Experiments

We performed several experiments to validate our GPU synthetic benchmark generation framework MINIME-GPU. MINIME-GPU and all of our benchmarks can be downloaded from our website[3] . We set the overall similarity score threshold as 90%, the individual similarity score threshold as 60%, and iteration upper bound to 20. These similarity scores are the maximum achievable scores with our framework. Note that we performed all experiments using Multi2Sim except the real hardware experiments in Section 5.5.5.

Table 5.2 shows the platforms that we run our experiments on. Unless otherwise specified we perform all experiments except the ones in the assessing architecture changes section on AMD HD 7970 GPU platform. Note that we target AMD SI GPUs because the current version of Multi2Sim simulator does not fully support other GPU models (AMD Evergreen GPU and NVIDIA Fermi GPU). However, in principle, our approach can be applied to other GPUs as well. In the next section, we give details for benchmarks and tools we used.

[3]http://depend.cmpe.boun.edu.tr/tools/minimegpu

Table 5.2. GPU architectural configurations.

Platform	Compute Unit		Memory Hierarchy		
Device	Compute Units (CUs)	Processing Elements per CU	Register file size per CU	Local memory size per CU	Global memory size
HD 7970	32	64	256KB	64KB	3GB
HD 7870	20	64	256KB	64KB	2GB
HD 7850	16	64	256KB	64KB	2GB
HD 7770	10	64	256KB	64KB	1GB
HD 7870d	20	64	64KB	16KB	2GB
HD 7850d	16	64	128KB	32KB	2GB
HD 7770d	10	64	512KB	128KB	1GB

5.5.1. Simulation and Benchmarks

We performed experiments on a system running Ubuntu Linux 12.04 x64. We use Multi2Sim simulator 4.2, which is a cycle based detailed simulation framework for CPU-GPU heterogeneous computing, to run original applications and synthetic benchmarks. Multi2Sim is a fully configurable open source simulator that supports several CPU and GPU architectures such as x86 CPU and AMD Southern Islands GPU. Since we cannot gather all of the characteristics we described above with default Multi2Sim tool, we have added new extensions to the tool to gather our missing characteristics. For example, we capture and dump work dimension and global/local size characteristics in order to mimic work-items per work-group characteristic. Target architectures can be configured via configuration files and a user can create a new configure file or can select an existing one depending on the target platform. We also use the integrated Multi2C, the kernel compiler, to produce kernel binaries for synthetic benchmarks. Note that we only use the AMD Southern Islands architecture in the experiments since Multi2C fails to generate AMD Evergreen architecture binaries for synthetic benchmarks. We produce host binaries for synthetic benchmarks using GCC 4.6.3. We

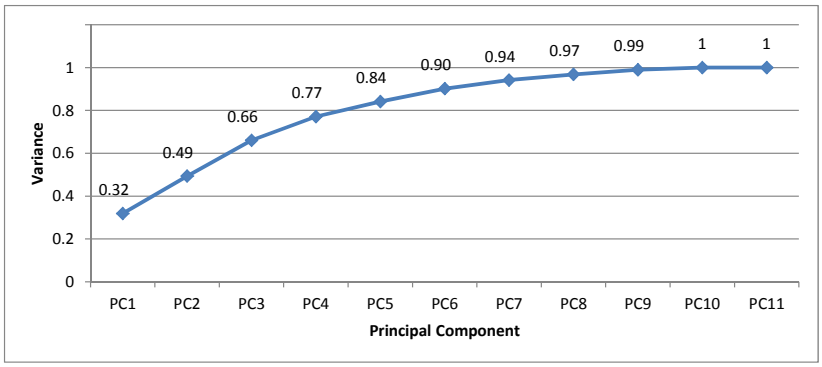

Figure 5.7. Principal components and their variance.

use AMD Catalyst 13.20 driver and AMD APP SDK v2.9 with OpenCL 1.2 during generation of synthetic benchmarks. We compiled synthetic benchmarks with '-O0' option so that the compiler did not remove our code blocks. Hence, our synthetic benchmarks cannot be used in compiler optimization studies.

We run experiments on all 23 benchmarks from AMD APP SDK 2.5 provided with Multi2Sim. These are the only GPU benchmarks that run on Multi2Sim. We used default (medium) inputs to generate synthetic benchmarks. We also used small and large inputs to assess input dependence of synthetic benchmarks. Note that we cannot use benchmarks from other benchmark suites including Rodinia [4] and Parboil [5] because they are not supported by the current distribution of Multi2Sim. Once Multi2Sim supports collecting characteristics of benchmarks from these suites, we are planning to extend our experiments on these suites.

5.5.2. Applying PCA to Validate the Importance of Characteristics

After the characterization of an original GPU application, we gather a set of characteristics and generate a data set. It is important that each characteristic in this data set contributes to the behavior of the application. Hence, we perform PCA statistical analysis [33] on all original benchmarks for all (eleven) characteristics described above to select the most suitable characteristics for benchmark generation. We use MATLAB [53] and appropriate libraries to implement PCA.

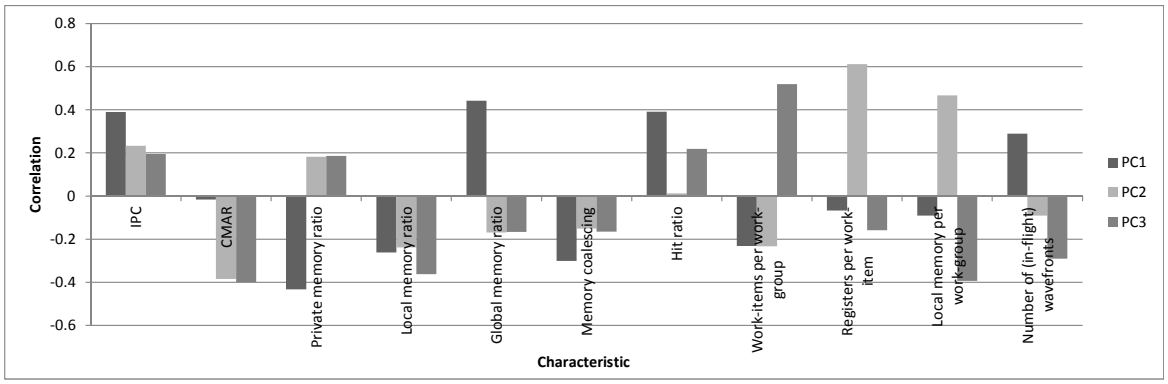

Figure 5.8. PCA loadings with respect to each characteristic.

Typically, most of the variance is contained in the first two or three principal components (PCs). However, in our case, we need to use the first seven PCs that capture more than 90% of the total variance of our data set as shown in Figure 5.7. We show factor loadings for the first three (PCs) in Figure 5.8. We use only the first three PCs since they capture nearly 70% of the total variance of our data set and the other PCs capture a small amount of the total variance. Note that a factor loading closer to -1 or +1 indicates a higher influence on the principal component. We observe that instruction throughput, memory instruction mix, and memory efficiency have the highest correlation with PC1, compute unit occupancy and computation-to-memory access ratio have the highest correlation with PC2, and compute unit occupancy has the highest correlation with PC3. These correlations validate that each characteristic we gather is important to describe the behaviors of an application.

5.5.3. Synthetic Benchmark Generation Results

Table 5.3 shows our synthetic benchmark generation results on AMD HD 7970 GPU platform. In the table, we show *dwarf* type [54] and the number of instructions (*#OrgInst*) for each original application (benchmark). Note that a dwarf defines an algorithmic method that captures computations and communication patterns of an application. Hence, benchmarks with different dwarfs have different behaviors and we validate that our approach can work on a diverse set of applications. We denote the number of iterations that it takes to generate a synthetic benchmark by *#iter*. In many cases, we generate synthetic benchmarks in less than 10 iterations where the

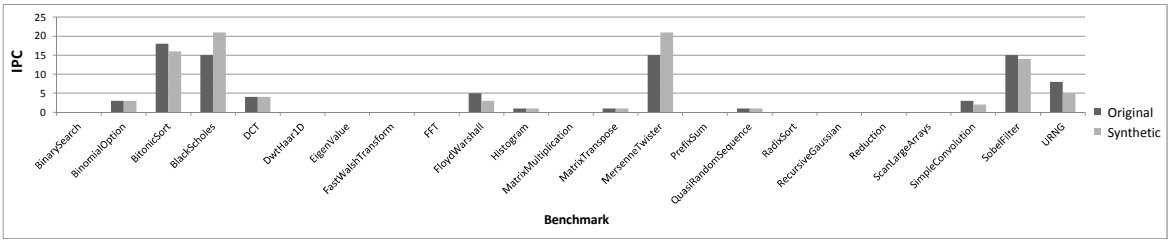

Figure 5.9. Comparison of IPC between the synthetic and original benchmarks.

maximum number of iterations is 15 for `MatrixTranspose`. We use *Speedup(×)* to denote the ratio of the GPU simulation time of the original application to the GPU simulation time of the corresponding synthetic benchmark. On average, our approach speeds up GPU simulation by a factor of 513×, 539×, 550×and 567× for HD 7970, HD 7870, HD 7850, and HD 7770 platforms, respectively. The harmonic mean speedup is 541×. The minimum speedup is 1.1× for `URNG`, and the maximum speedup is 7284.8× for `EigenValue`. We observe that obtaining a higher speedup for small applications is difficult because they are already fast with a small number of instructions. Also, when an original application has a characteristic with a very low or high value, the size of our code blocks increase, hence we have a low speedup value. For example, CMAR of original `MersenneTwister` is very low (0.2) and we add CB4 to decrement CMAR of the synthetic benchmark where *LOOP_COUNT* is 27. Since we have a high *LOOP_COUNT* value, the dynamic instruction count of the synthetic is also high, which results in a low speedup. Furthermore, when the work-group count of a synthetic benchmark, which is the total number of work-groups executed, is high the number of instructions is also high, hence, the speedup is low. For example, `SobelFilter` and `URNG` have 1024 and 4096 work-groups and their speedups are 1.3× and 1.1×, respectively. In the table, we show the overall similarity score (in the column *OSS*) for each benchmark where the average overall similarity score is 96%, the minimum overall similarity score is 92% for `URNG` and the maximum overall similarity score is 100% for `EigenValue` and `MatrixMultiplication`.

Table 5.3. Synthesis results on AMD HD 7970 platform.

Benchmark	**Dwarf**	**#OrgIns**	**#iter**	**Speedup(×)**	**OSS**
BinarySearch	Graph Traversal	128	2	2.8	95
BinomialOption	Dense Linear Algebra	1984512	4	516.2	95
BitonicSort	Graph Traversal	1267200	4	44.5	95
BlackScholes	Sparse Linear Algebra	1139712	8	3.4	94
DCT	Spectral Methods	20672	9	3.6	96
DwtHaar1D	Spectral Methods	2113	10	1.4	97
EigenValue	Spectral Methods	19826681	6	7284.8	100
FastWalshTransform	Dense Linear Algebra	4000	4	8.0	99
FFT	Spectral Methods	1316	8	14.1	98
FloydWarshall	Dynamic Programming	11046882	2	280.9	90
Histogram	Structured Grids	334912	3	190.1	96
MatrixMultiplication	Sparse Linear Algebra	8000	6	11.1	100
MatrixTranspose	Dense Linear Algebra	3392	15	1.2	96
MersenneTwister	Combinational Logic	1347584	8	4.0	94
PrefixSum	Dense Linear Algebra	1341	10	2.6	97
QuasiRandomSequence	Combinational Logic	32288	11	6.6	98
RadixSort	Graph Traversal	39488	3	2244.0	99
RecursiveGaussian	Dense Linear Algebra	1606368	10	1159.0	99
Reduction	Dense Linear Algebra	714	7	4.3	95
ScanLargeArrays	Backtrack & Branch+Bound	3688	5	8.3	93
SimpleConvolution	Structured Grids	17900	6	2.0	94
SobelFilter	Dense Linear Algebra	1297944	7	1.3	93
URNG	Structured Grids	2146304	11	1.1	92

5.5.4. Assessing Similarity

Now, we demonstrate individual similarity scores for each characteristic. Figure 5.9 shows IPCs for the synthetic and original benchmarks where the average IPC similarity score is 91%, the minimum IPC similarity score is 60% for only three benchmarks and the maximum IPC similarity score is 100% for sixteen benchmarks. Note that Multi2Sim captures IPC values as integer values and rounds them down to 0 when they are smaller than 1. In the figure, the IPCs of `BlackScholes` and its corresponding synthetic benchmark are 15 and 21, respectively, hence the IPC similarity score

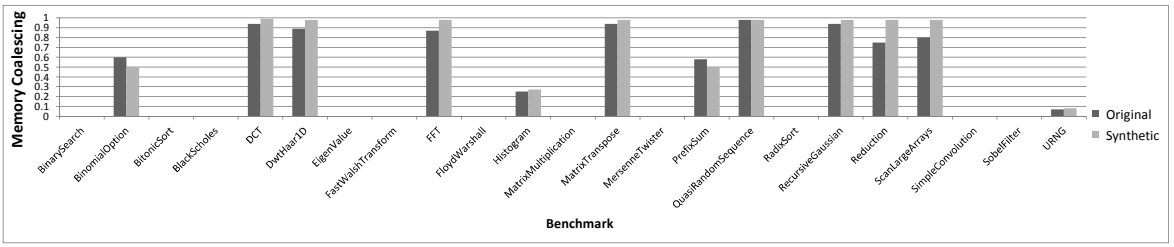

Figure 5.10. Comparison of memory coalescing between the synthetic and original benchmarks.

is 60%. We observe that once we add code blocks that increment private memory ratio and decrement CMAR of the synthetic, we meet the overall (90%) and individual (60%) similarity score thresholds. We perform a new experiment in which we set the individual similarity score threshold to 80% to check whether we can improve the IPC similarity for `BlackScholes`. In this case, we observe that adding a code block to improve IPC similarity has a side effect on CMAR similarity and decreases it. We observe that the similarity score thresholds (90% and 60%) are the maximum achievable scores with our framework due to side effects and we are planning to handle these side effects as a future work. Due to side effects, the original and the synthetic benchmark can diverge and there is no systematic bias for similarity scores of the synthetic benchmarks. Note that handling side effects in C is harder than handling them in assembly.

Figure 5.10 shows memory efficiencies for the synthetic and original benchmarks where the average memory coalescing similarity score is 94%, the minimum memory coalescing similarity score is 69% for only one benchmark and the maximum memory coalescing similarity score is 100% for twelve benchmarks. We display the results for IPC and memory coalescing but we had similar results for other characteristics as well.

5.5.5. Validation of Synthetic Benchmarks on Real Hardware

In this section, we validate the robustness of the synthetic benchmarks generated on the simulator platform by running them and the original applications on an actual hardware and checking their similarity. In these experiments, we use the source code of the synthetic benchmarks that are generated on the simulator for AMD HD 7970

GPU and run these synthetic benchmarks on a real AMD HD 7950 GPU hardware (without re-generation). AMD HD 7950 GPU is similar to 7970 GPU except that it has 28 compute units instead of 32.

We performed experiments on an HP Z800 Desktop Workstation system running Windows 7 SP1 x64. We compiled host binaries for synthetic benchmarks using Microsoft Visual Studio 2010. We use AMD Catalyst 14.12 driver and AMD APP SDK v2.9 with OpenCL 1.2 during the generation of kernel code of the synthetic benchmarks. We use '-cl-opt-disable' option, which disables all optimizations, so that the compiler did not remove our code blocks. We used AMD CodeXL 1.7 tool [55] to collect performance characteristics of original and synthetic benchmarks on the real hardware. CodeXL is the most commonly used tool in the literature for collecting OpenCL program characteristics on AMD GPUs.

We observe that our synthetic benchmarks mimic all but IPC and memory coalescing characteristics of original applications on the real hardware. We were unable to check IPC and memory coalescing because CodeXL does not provide these characteristics. However, CodeXL provided other characteristics that were not available in Multi2Sim. These characteristics include VALU (Vector Arithmetic Logic Unit) utilization and SGPRs (Scalar General-Purpose Registers). Figure 5.11 shows VALU utilization characteristic, which is the percentage of active vector ALU threads in a wave, for the synthetic and original benchmarks where the average VALU utilization similarity score is 83%. The maximum VALU similarity score is 100% for twelve benchmarks. The minimum similarity score is 0% for `DwtHaar1D` where the VALU utilization of the original benchmark is 39% and the VALU utilization of the corresponding synthetic benchmark is 100%. This is because we cannot collect the registers per work-item characteristic due to a bug in Multi2Sim, hence we cannot preserve the compute unit occupancy characteristic of the original benchmark which influences VALU.

Figure 5.12 shows SGPRs characteristic, which is the number of SGPRs used by the kernel, for the synthetic and original benchmarks where the average SGPRs similarity score is 73%. The maximum similarity score is 100% for `Histogram` and the

minimum similarity score is 50% for `BlackScholes` and `DwtHaar1D`.

We observe that the average cache hit ratio similarity on HD 7950 is 68%, whereas it is 82% on HD 7970. Note that cache hit ratio (provided by CodeXL) is the percentage of fetch, write, atomic, and other instructions that hit the data cache and this characteristic is different from the one we used in Multi2Sim. There exist several reasons behind this observation. The AMD Catalyst driver version that Multi2Sim supports is version 13.20, whereas CodeXL supports version 14.12. Hence, different drivers can generate different OpenCL binaries. Another reason is the estimation error of Multi2Sim simulator which can be more than 30% due to the lack of fidelity in the way the memory subsystem is simulated.

We also observe that our synthetic benchmarks mimic the MemUnitStalled, WriteUnitStalled, and LDSBankConflict characteristics of the original benchmarks where MemUnitStalled is the percentage of GPU time the memory unit is stalled, WriteUnitStalled is the percentage of GPU time the write unit is stalled, and LDSBankConflict is the percentage of GPU time Local Data Storage (LDS) is stalled by bank conflicts. The average MemUnitStalled, WriteUnitStalled, and LDSBankConflict similarity between the original and synthetic benchmarks is 65%, 73%, and 85%, respectively. Note that we do not use these characteristics during benchmark generation, hence the similarity score can be low (e.g. 65%). One can use these characteristics of an original application in the synthetic benchmark generation process to improve the similarity but this will result in runtime penalty.

Although our synthetics do not include code blocks for the characteristics including VALU utilization and SGPRs they are still preserved. This further confirms the robustness of our synthetic benchmarks. In order to improve the similarity between synthetic and original benchmarks in real hardware environment, we can generate synthetics in real hardware environment instead of a simulator environment similar to our technique for CPUs.

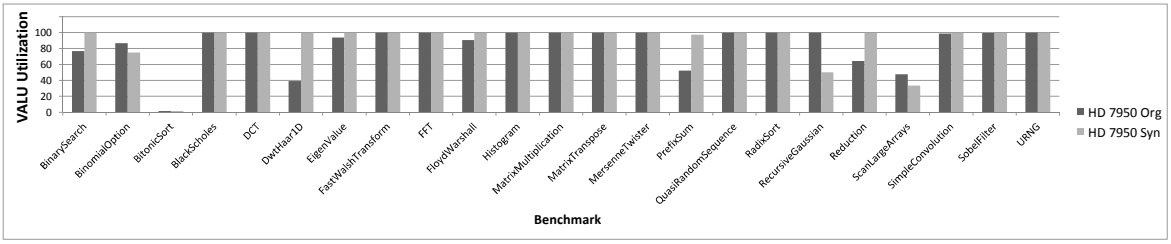

Figure 5.11. Comparison of VALU utilization between the synthetic and original benchmarks on real hardware (HD 7950).

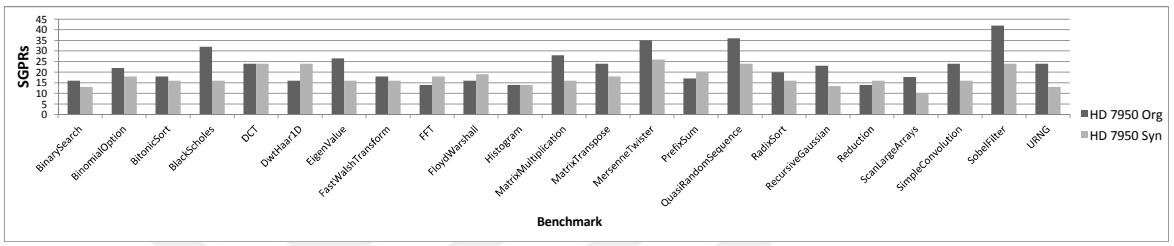

Figure 5.12. Comparison of SGPRs utilization between the synthetic and original benchmarks on real hardware (HD 7950).

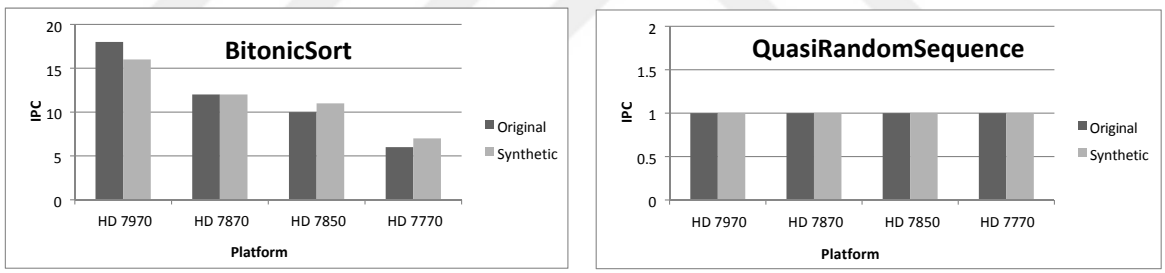

Figure 5.13. Comparison of sensitivity to architecture changes for `BitonicSort` and `QuasiRandomSequence`.

5.5.6. Assessing Architecture Changes

When developing synthetic benchmarks, it is important that synthetic benchmarks are portable, that is, they preserve the behaviors of original applications they are generated from across different architectures. This is because we want benchmarks to allow architectural exploration. In order to assess the usage of our synthetic benchmarks on different architectures, we perform a set of experiments on four different existing platforms, which are AMD HD 7970, HD 7870, HD 7850 and HD 7770. We show these platforms, which have different configurations, in Table 5.2. In the experiments, we generated synthetic benchmarks on AMD HD 7970 GPU and then ran these

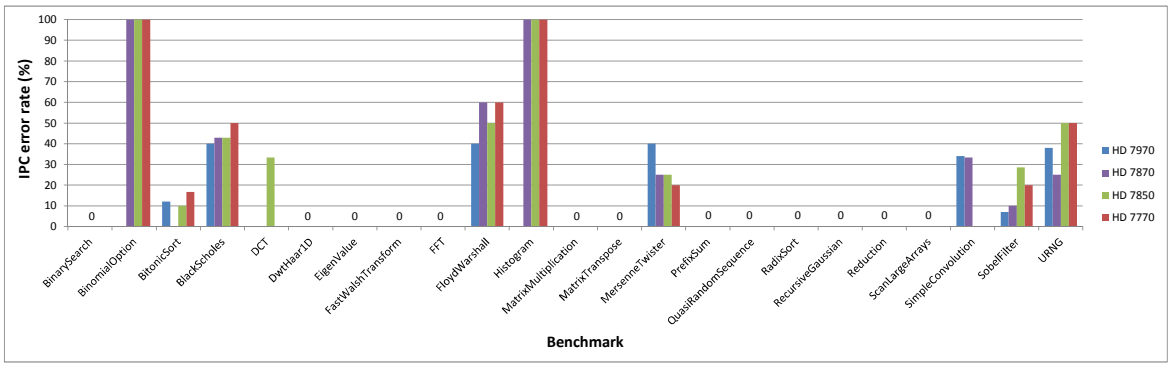

Figure 5.14. The IPC error score for different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770).

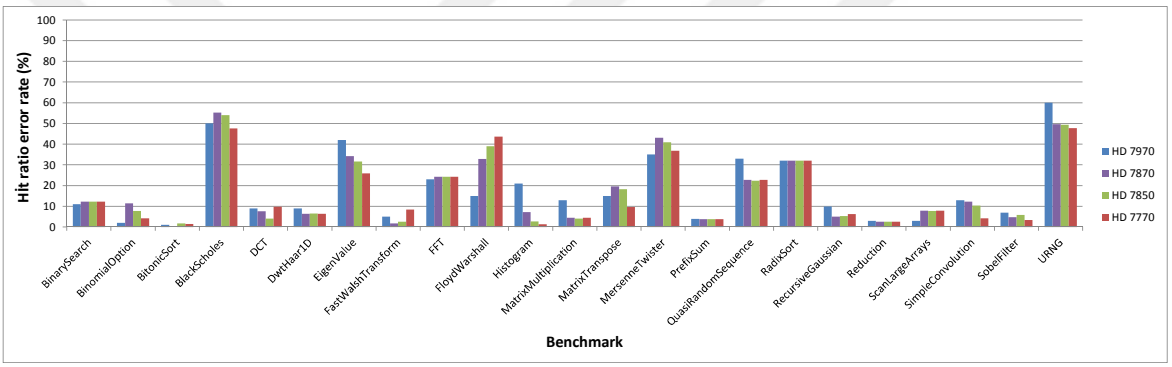

Figure 5.15. The hit ratio error score for different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770).

synthetic benchmarks on other platforms (without re-generation). We observe that our synthetic benchmarks mimic the behaviors of original applications across different platforms. For example, we show comparison of sensitivity to architecture changes for `BitonicSort` and `QuasiRandomSequence` in Figure 5.13. The IPC of `BitonicSort` benchmark decreases going from HD 7970 (on which the synthetic benchmark is generated) to other platforms and the IPC of the synthetic benchmark follows this change. The IPC of `QuasiRandomSequence` does not change going from HD 7970 to other platforms and similarly, the IPC of the synthetic benchmark does not change. The correlation coefficients for `BitonicSort` and `QuasiRandomSequence` are 0.99 and 1, respectively and the average of the correlation coefficient for all benchmarks is 0.93.

Figure 5.14 shows the IPC error score for all platforms and Figure 5.16 shows IPCs of the original and synthetic benchmarks. From the figure, it is clear that our

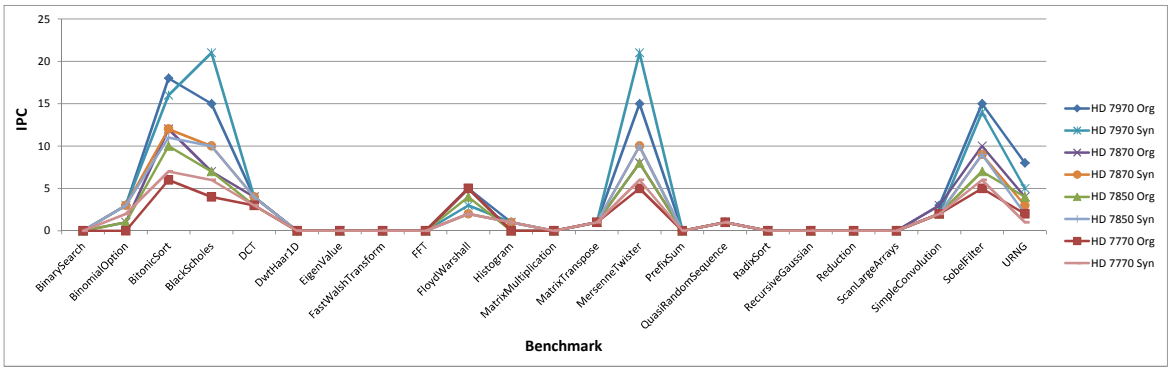

Figure 5.16. IPCs of original and synthetic benchmarks on different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770).

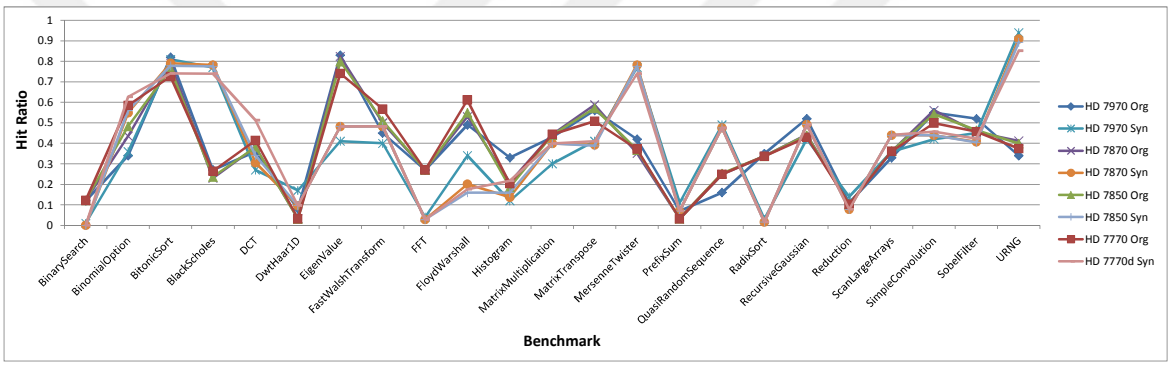

Figure 5.17. Hit ratios of original and synthetic benchmarks on different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770).

synthetic benchmarks mimic the behavior changes of the original benchmarks across different platforms. We observe that `BinomialOption` and `Histogram` benchmarks have high IPC error scores moving from HD 7970 to other platforms. This is because the IPC of original benchmarks are very low (near to zero) and a small difference between the IPC of the original benchmark and the synthetic benchmark results in a high error score. For instance, the IPCs of the original and synthetic `Histogram` benchmarks is 1 on HD 7970 platform where the error score is 0%. When we move the original benchmark to other platforms, the IPC becomes 0. However, the IPC of the corresponding synthetic benchmark remains 1. Hence, this small difference (1, in terms of IPC) results in a high (100%) error score. Figure 5.15 shows the hit ratio error score for all platforms and Figure 5.17 shows hit ratios of the original and synthetic benchmarks. Note that although we do not use hit ratio characteristic, which is the average of L1 cache, L2 cache, and global memory hit ratio, in similarity measurement

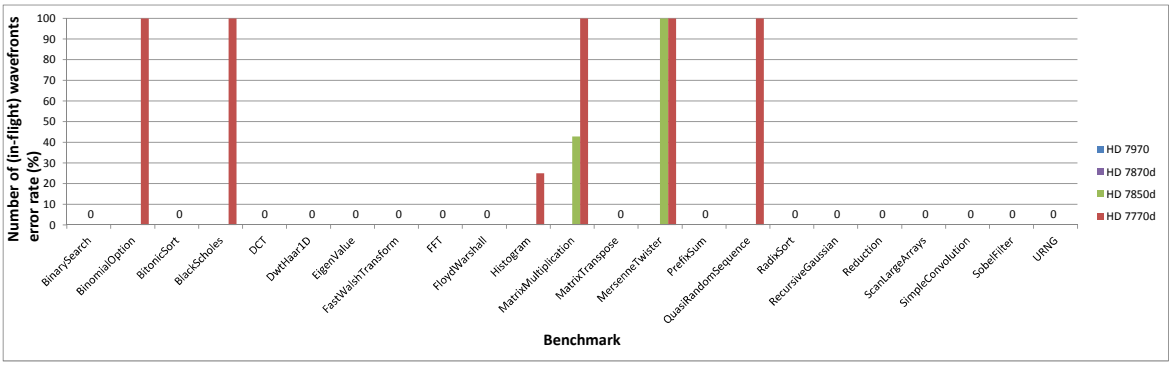

Figure 5.18. The number of (in-flight) wavefronts error score for the GPUs having derived architectural configurations (HD 7970, HD 7870d, HD 7850d, and HD 7770d).

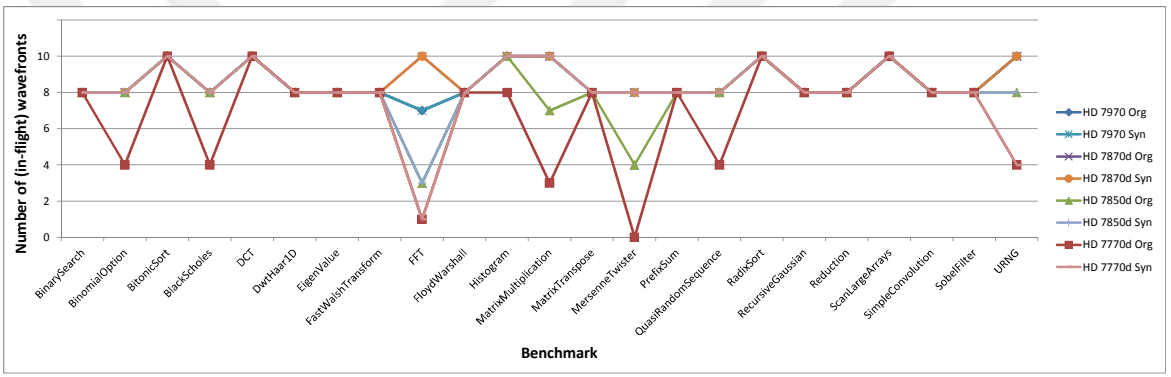

Figure 5.19. Number of (in-flight) wavefronts of original and synthetic benchmarks on the GPUs having derived architectural configurations (HD 7970, HD 7870d, HD 7850d, and HD 7770d).

during benchmark generation, the average hit ratio similarity between the original and synthetic benchmarks on HD 7970 is 82% and our synthetic benchmarks mimic the hit ratio changes of the original benchmarks across different platforms. We also observe that our synthetic benchmarks mimic the L1, L2, and global memory hit ratio changes of the original benchmarks across different platforms where the average L1, L2 and global memory hit ratio similarity between the original and synthetic benchmarks on HD 7970 is 84%, 70%, and 98%, respectively.

5.5.6.1. Synthesis for GPUs having Derived Architectural Configurations. We observe that the only difference between the existing four platform configurations is the number of compute units. Hence, we perform a new set of experiments on platforms that we derived from the existing platforms by changing register file size and local memory size

per CU. Table 5.2 shows these platforms (HD 7870d, HD 7850d and HD 7770d), which are not part of Multi2Sim distribution. Global memory sizes were not changed since the memory demands of the benchmarks that we use are small. In the experiments, we run the synthetic benchmarks generated on HD 7970 on the other platforms (without re-generation). We observe that CMAR, private memory ratio, local memory ratio, global memory ratio, memory coalescing, work-items per work-group, and local memory per work-group characteristics are platform independent and they do not change across platforms as expected. IPC and hit ratio characteristics change only when the number of compute units changes. For example, these characteristics do not change moving from HD 7870 to HD 7870d but, they change moving from HD 7970 to HD 7870 and our synthetic mimics these changes as shown in Figures 5.14 and 5.15.

The number of (in-flight) wavefronts characteristic depends on work-items per work-group, registers per work-item, and local memory per work-group characteristics. Our experiments demonstrate that number of (in-flight) wavefronts characteristic changes moving from HD 7970 to a derived platform due to changes in registers per work-item and local memory per work-group characteristics. Figure 5.18 shows the numbers of (in-flight) wavefronts error score for all derived platforms and Figure 5.19 shows numbers of (in-flight) wavefronts of the original and synthetic benchmarks. For example, the number of (in-flight) wavefronts of `FFT` benchmark on HD 7970, HD 7870d, HD 7850d, and HD 7770d, is 7, 10, 3, and 1, respectively and the corresponding synthetic benchmark mimic these values on each platform. Hence, the error scores are 0% on all platforms. We observe that the local memory per work-group limits the number of (in-flight) wavefronts for `FFT`. Hence, increasing the local memory size per compute unit increases the number of (in-flight) wavefronts and we mimic the number of (in-flight) wavefronts for `FFT` since we mimic the local memory size per compute unit. However, the number of (in-flight) wavefronts of `MatrixMultiplication` benchmark on HD 7970, HD 7870d, HD 7850d, and HD 7770d, is 10, 10, 7, and 3, respectively and it is 10 for the corresponding synthetic benchmark on all platforms. Hence, the error scores are 0%, 0%, 43%, and 100%, respectively. Note that we cannot mimic the changes across different platforms since registers per work-item limits the number of (in-flight) wavefronts for `MatrixMultiplication` that results in a high error score, and

Table 5.4. GPU architectural parameters.

Parameter	HD 7970	HD 7970d1	HD 7970d2	HD 7970d3
Front-end issue latency	1	2	4	8
Front-end issue width	5	8	12	16
SIMD unit width	1	2	4	8
Scalar unit width	1	2	4	8
Scalar unit ALU latency	4	8	12	16
Vector memory unit width	1	2	4	8
Branch unit width	1	2	4	8

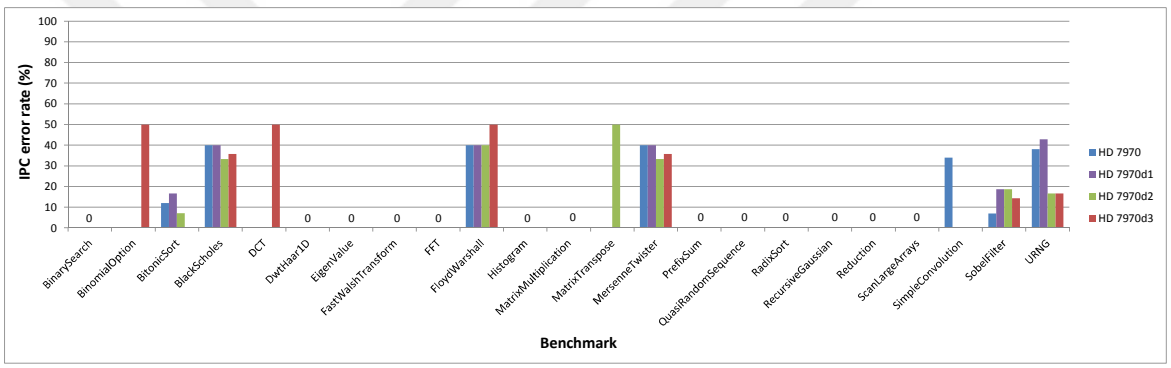

Figure 5.20. The IPC error score for the GPUs having derived architectural parameters (HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3).

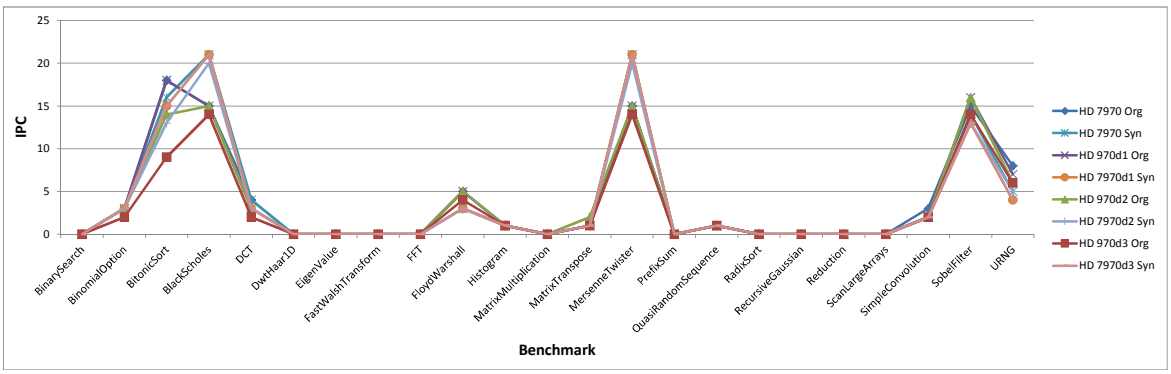

Figure 5.21. IPCs of original and synthetic benchmarks on the GPUs having derived architectural parameters (HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3).

we cannot mimic registers per work-item characteristic. Once the bug in Multi2Sim is fixed, we can also mimic registers per work-item characteristic as well as number of (in-flight) wavefronts characteristic.

We also perform a new set of experiments on platforms that we derived from the existing HD 7970 platform, which are denoted by HD 7970d1, HD 7970d2 and HD 7970d3. We show these platforms, which have different architectural parameters including front-end issue latency, SIMD unit width, and scalar unit ALU latency, in Table 5.4. In the table, *front-end issue latency* is the number of cycles that it takes to issue a wavefront to its execution unit and *front-end issue width* is the maximum number of instructions that can be executed in a single cycle. *SIMD unit width* is the number of instructions processed by each stage of the pipeline per cycle. *Scalar unit width*, *vector memory unit width*, and *branch unit width* are similar to *SIMD unit width*. *Scalar unit ALU latency* is the number of cycles it takes to execute a scalar arithmetic logic instruction. In the experiments, we run the synthetic benchmarks generated on HD 7970 on other platforms derived from HD 7970 (without re-generation). We observe that IPC changes moving from HD 7970 to other derived platforms and our synthetic mimics these changes as shown in Figure 5.20 where the IPC similarity score is 91%, 91%, 91%, and 89% on HD 7970, HD 7970d1, HD 7970d2 and HD 7970d3, respectively. Also, we show IPCs of the original and synthetic benchmarks in Figure 5.21. For example, the IPC of original `Histogram` benchmark is 1 on all platforms and similarly, the IPC of the synthetic benchmark is 1 on all platforms. On the other hand, the IPC of original `BitonicSort` benchmark decreases moving from HD 7970 to other platforms and the corresponding synthetic benchmark mimics these changes. That is, the IPC of original `BitonicSort` benchmark is 18, 18, 14, and 9 and the IPC of the corresponding synthetic benchmark is 16, 15, 13, and 9 on HD 7970, HD 7970d1, HD 7970d2 and HD 7970d3, respectively.

Next, we considered other characteristics including memory instruction mix and hit ratio to evaluate the portability of our synthetic benchmarks. We find the synthetic benchmarks to accurately mimic these characteristics compared to the original benchmarks across GPUs having different architectural parameters.

Also, we perform a new set of experiments on HD 7970 platforms with different cache configurations. We show these cache configurations in Table 5.5. In the experiments, we run the synthetic benchmarks generated on HD 7970 with *Config-0*

Table 5.5. GPU cache configurations.

Configuration	Config-0	Config-1	Config-2	Config-3
L1 - Sets	64	128	32	64
L1 - Associativity	4	4	4	2
L2 - Sets	128	256	64	128
L2 - Associativity	16	16	16	8

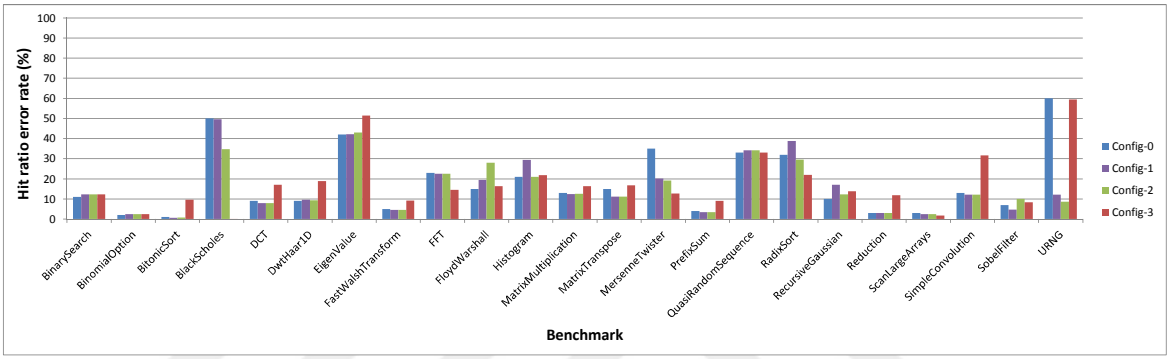

Figure 5.22. The cache hit ratio error score for different cache configurations (*Config-0*, *Config-1*, *Config-2* and *Config-3*).

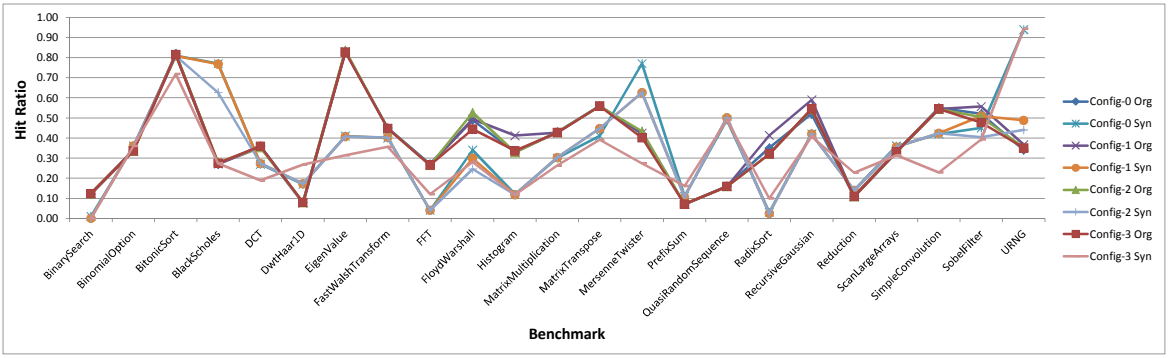

Figure 5.23. Cache hit ratios of original and synthetic benchmarks on different cache configurations (*Config-0*, *Config-1*, *Config-2* and *Config-3*).

on other platforms with different cache configurations (without re-generation). We observe that the cache hit ratio changes moving from *Config-0* to other configurations and our synthetic mimics these changes as shown in Figure 5.22 where the average hit ratio similarity score is 82%, 84%, 85%, and 82% on *Config-0*, *Config-1*, *Config-2* and *Config-3*, respectively. Also, we show hit ratios of the original and synthetic benchmarks in Figure 5.23. For example, the hit ratio of original `BinomialOption`

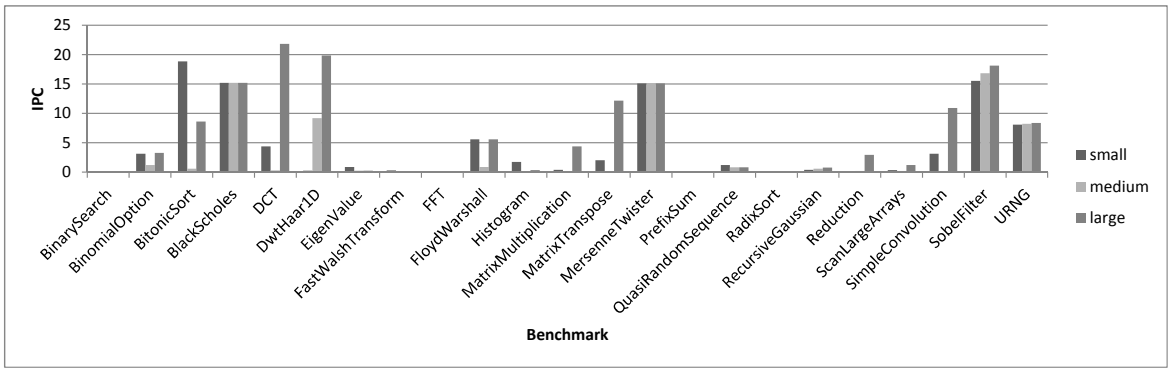

Figure 5.24. The IPC values of original benchmarks for small, medium, and large inputs on HD 7970.

benchmark is 36% on all configurations and similarly, the hit ratio of the synthetic benchmark is 34% on all configurations. On the other hand, the hit ratio of original `SobelFilter` benchmark changes moving from *Config-0* to other configurations and the corresponding synthetic benchmark mimics these changes. That is, the hit ratio of original `SobelFilter` benchmark is 52%, 56%, 50%, and 48% and the cache hit ratio of the corresponding synthetic benchmark is 45%, 51%, 40%, and 39% on *Config-0*, *Config-1*, *Config-2* and *Config-3*, respectively.

5.5.7. Assessing Input Changes

In these experiments, we analyze whether we can use the synthetic benchmark for an original application using a different input than the one for which the synthetic benchmark was generated for. In this analysis, we generate a synthetic benchmark for an original benchmark using a medium input. Then, we measure the similarity between this synthetic benchmark and the original benchmark using a different (small or large) input. If the characteristics of the original application do not vary much from using medium input to a different input, that is, the similarity meets the user (individual and overall) thresholds, we can use the same synthetic benchmark for the original benchmark using a different input. Otherwise, we need to generate a new synthetic benchmark. Note that this analysis helps to reduce the effort of generating new synthetic benchmarks.

Figure 5.24 shows the IPC values of original benchmarks for small, medium, and large inputs on HD 7970. We observe that some benchmarks have similar IPC values for different inputs and some benchmarks have different IPC values. For example, we need to generate a new synthetic benchmark for `DwtHaar1D` using small (and also for large) input since individual IPC similarity score is less than the individual similarity threshold. However, the IPC of `URNG` does not vary much from using medium input to small (and also large) input, hence, we can use the same synthetic benchmark for `URNG` using small, medium, and large input. Similarly, we measure the individual similarity scores for other characteristics and overall similarity rate and we decide whether we can use the existing synthetic benchmark or we need to generate a new one.

5.6. Discussion

We observe that the main cost of generating synthetic GPU benchmarks is collecting the characteristics of original applications. This is because we use a simulator (Multi2Sim) to collect these characteristics and simulating large (original) applications takes long time. On the other hand, our synthetic benchmarks are small and simulating these synthetic benchmarks is fast. For example, collecting the characteristics of `FloydWarshall` benchmark takes 767.04 seconds, however, the corresponding synthetic benchmark is generated in 2 iterations and collecting the characteristics of the synthetic benchmark takes only 1.47 seconds in the first iteration and 9.40 seconds in the second iteration. Note that characteristics collection in the initial iterations during benchmark generation is faster since the number of code blocks can increase with iterations.

It is clear that the speedup we obtain by using a synthetic benchmark instead of an original application is more important than the time required to generate the synthetic benchmark. This is because we generate a synthetic benchmark only once and we run this synthetic benchmark many times. For example, generating the synthetic benchmark for `FloydWarshall` takes nearly 780 seconds but we obtained a synthetic benchmark that is 431.29× faster than the original benchmark.

When generating synthetic benchmarks, selecting the right characteristics to mimic is crucial for the time required to generate the synthetic benchmark, for speedup, and the similarity score. For example, we performed a set of experiments in which we only mimic the IPC of original applications. We observed that the average IPC similarity goes from 91% to 94%, the average speedup goes from 513× to 56438×, and the minimum IPC similarity goes from 60% to 66% on HD 7970 platform comparing to using eleven characteristics. These results validate that using less number of characteristics provide higher similarity as well as higher speedup. This is because using too many characteristics can result in a long characterization process as well as less similar synthetic benchmarks since adding a code block to improve one characteristic similarity can decrease another characteristic similarity due to side effects. On the other hand, using too few characteristics can fail to mimic the behavior of an original application. In our approach, we use eleven characteristics that capture the diverse behaviors of an original application. We also experimentally validate that many of these characteristics are platform independent and using platform independent characteristics makes our benchmarks portable across a wide range of platforms as shown in the experiments.

Another important point when generating synthetic benchmarks is that the synthetic benchmarks can be used for a study where they were not originally intended to be used. In order to validate this case, we performed a set of experiments. In these experiments, we compared IPC similarity when it was used during synthesis and when it was not used during synthesis. We observed that the average IPC similarity goes from 91% to 84%, the average speedup goes from 513× to 665×, and the minimum IPC similarity goes from 60% to 50% on HD 7970 platform when IPC similarity is used and not used during synthesis, respectively. These results show that average IPC similarity is still high and acceptable.

Our synthetic benchmarks do not have any useful functionality and one cannot obtain the original application from our synthetic benchmarks by reverse engineering. Hence, customers can share a synthetic benchmark that is generated for their proprietary application without compromising on the proprietary nature of the proprietary

application. Once the hardware developers, architects, or designers have the synthetic benchmark, they can optimize the (GPU) platform to provide improved performance for the proprietary application.

Our synthetic benchmarks are meant to be used in early design exploration. In later stages of development where more accurate performance is required original applications are still going to be used. Also, when using our synthetics, one should note the characteristics that are kept similar in synthetics with respect to the originals and use synthetics for early design exploration of such characteristics.

5.7. Summary

We developed a new benchmark synthesis framework for GPUs, called MINIME-GPU, to speed up architectural simulation of modern GPU architectures. Our framework captures important characteristics of original GPU applications and generates synthetic GPU benchmarks from those applications. We compared the similarity (accuracy) of original existing applications and the corresponding synthetic benchmarks in terms of several characteristics including instruction throughput, compute unit occupancy, and memory efficiency. The experimental results showed that our synthetic benchmarks mimic the characteristics of the original applications they are generated from where the average similarity is 96% and average speedup is 541×. Also, we experimentally validated that our synthetic benchmarks preserve these characteristics across different architectures on a simulator as well as on a real GPU. Our synthetic benchmarks are generated in OpenCL, which is portable, human readable, and widely used for GPU programming, and they are also faster and smaller than the original applications. Hence, our framework helps developers and researchers in performance analysis and early architectural exploration of software and hardware.

6. USING MACHINE LEARNING TECHNIQUES TO DETECT PARALLEL PATTERNS OF MULTI-THREADED APPLICATIONS

6.1. Overview

In this chapter, we show how machine learning techniques can be effectively used to automatically detect parallel patterns in multi-threaded applications as a guideline for improvements. Machine learning techniques are often feasible and cost-effective for classification and they are widely used in business, science, industry, and government [13, 14]. Hence, in order to overcome the cost of manual parallel pattern detection, we develop a machine leaning framework that *automatically* classifies multi-threaded applications based on their parallel patterns. We implemented several machine learning techniques for this purpose: k-nearest neighbor, decision trees, naive Bayes classifier, neural networks, and principal component analysis. We also implemented a feature selection technique to improve classification results.

Parallel pattern detection begins with the characterization of the given multi-threaded application. Characterization involves describing an application as a set of quantifiable attributes. Since parallel patterns are high level characteristics, we use high level thread communication and data sharing behaviors to describe the software architectural characteristics of an application. Whereas, low level microarchitecture dependent characteristics including CPI, cache miss rate, branch misprediction rate are commonly used in the literature [56–58] where they capture performance characteristics instead of software architectural characteristics. Note that these characteristics cannot be reliably obtained through static analysis but rather a dynamic analysis including binary instrumentation and profiling with hardware counters as is commonly used in the literature [3, 40]. Designers can then use these characteristics to understand the application's important features.

We experimentally validate that our parallel pattern detection framework can classify the benchmarks in well-known multicore benchmark suites, namely PARSEC and Rodinia. We compare techniques (machine learning models) in terms of speed and accuracy since different uses of parallel patterns have different requirements from the classification technique. For instance, dynamic thread mapping with parallel patterns as in [59] requires the classification technique to be fast and synthetic benchmark generation with parallel patterns as in Chapter 3 requires it to be accurate. Note that the speed is the amount of time needed for a machine learning to perform learning and classification and the accuracy is the percentage of instances in the test set that are correctly classified. We also show the usefulness of our parallel pattern detection framework for synthetic benchmark generation. Our experiments show that kNN, naive Bayes classifier, and decision trees correctly recognize parallel patterns of the benchmarks with a 100%, 96%, 92% accuracy, respectively. At the same time, the decision tree is the fastest technique with the lowest characterization overhead. On average, the decision tree results in a 5.7× classification speedup over the other techniques without feature selection and 4.8× speedup with feature selection. Our experiments also validate that the synthetic benchmarks generated by using parallel patterns are similar on average 92% to the original benchmarks, the average speedup is 23× and the average code reduction is 20×.

We first published our results on pattern detection in [26]. In particular, this chapter makes the following contributions.

- We apply machine learning techniques in a novel approach to automatically detect parallel patterns and we compare these techniques in terms of accuracy and speed.
- We implemented our machine learning techniques as a part of MINIME framework and experimentally validated the detection ability of them on benchmarks including PARSEC and Rodinia.
- We demonstrate that k-nearest neighbor, naive Bayes classifier, and decision trees are the most accurate techniques with a 100%, 96%, and 92% accuracy, respectively.
- We show that decision trees is the fastest technique where they provide a 5.7×

average characterization speedup over the other techniques that do not use feature selection and a 4.8× speedup over the other techniques that use feature selection.

- We also show the usefulness of the proposed techniques on synthetic benchmark generation.

6.1.1. Motivation for Classifying Parallel Patterns

Parallel patterns describe thread communication and data sharing behaviors of multi-threaded applications where these behaviors are important in many use cases including synthetic benchmark generation, dynamic task mapping and compiler optimization, parallel application development, and selecting an optimal architecture. Hence, automatically detecting parallel patterns by using a fast and accurate machine learning technique enables performance improvements and optimizations where other techniques including rewriting applications, determining compiler flag settings, and getting the next generation machine can be labor-intensive, time consuming, and costly as well as require expert knowledge. We describe parallel pattern use cases below. Note that all of the mentioned use cases have been validated in the literature either by us or by others.

- Synthetic benchmark generation: Synthetic benchmarks are artificial applications that mimic the characteristics of real-life applications. These benchmarks can be developed by varying application characteristics or they can be derived from existing applications such that they are faster and smaller than the existing applications. Hence, they can speed up the process of early performance evaluation and architectural exploration studies. Parallel patterns have been used to generate synthetic benchmarks from existing applications as in Chapters 3 and 4. In Section 6.4 (experiments section of this chapter), we show the usefulness of our proposed pattern recognition techniques.
- Dynamic task mapping and compiler optimization: Dynamic task mapping has been performed using the parallel pattern knowledge as in [59]. Their techniques improve existing dynamic mapping schemes by as much as 23%. The dynamic

task mapping approach necessitates a fast and low overhead pattern detection technique. Parallel patterns have been used to devise power optimization schemes in compilers by exploiting the opportunities of the recurring patterns of embedded multicore programs [60]. The authors validate their technique for low power with parallel design patterns on Finite Impulse Response (FIR) and image recognition applications and they observe significant power reduction.

- Parallel application development: Parallel patterns are used to organize development of parallel algorithms and programming models as in [61]. FastFlow [62], which is a structured parallel programming framework, simplifies parallel application development by providing a set of ready-to-use algorithmic skeletons that capture the most common parallel patterns. Also, in [63], the authors use parallel design patterns to develop efficient, maintainable, and portable parallel applications. Designers can develop parallel applications using refactoring approaches based on well-understood high-level parallel design patterns as in the ParaPhrase project [64] and [65]. In the Discovery of Potential Parallelism (DiscoPoP) project [66], different parallel patterns such as pipeline are detected to parallelize sequential programs.
- Selecting an optimal architecture: Parallel patterns have also been used for deciding the type of multicore architecture. For instance, in the ParaPhrase project [64, 67], the authors decide the required resources in heterogeneous systems using the parallel pattern of a target application. In general, heterogeneous multicore architectures including larger and smaller CPU cores are suitable for divide and conquer and recursive data patterns. This is because threads in these patterns are unbalanced and assigning larger threads to the larger cores and smaller threads to the smaller cores provides load balancing. On the other hand, homogeneous multicore architectures are suitable for geometric decomposition pattern, which has balanced threads.

6.2. Parallel Pattern Classification Using Machine Learning

We use machine learning techniques to automatically detect parallel patterns from an example set of multi-threaded applications to make parallel pattern classification more efficient. Note that using hard-coded manually derived heuristics instead of machine learning techniques requires experts to write rules by analyzing data. Moreover, when there exist enough number of training examples, machine learning techniques outperform hand-crafted solutions in terms of accuracy, speed, and scalability. For example, the accuracy of a speech recognition system is greater if one trains the system using machine learning than if one attempts to manually program it [68].

We use several machine learning techniques where each technique has advantages and disadvantages. For example, classification with decision trees is memory efficient and fast compared to the memory intensive k-nearest neighbor. Neural networks are slower than decision trees in both training and classification. Furthermore, neural networks do not present an easily-understandable classification model. Whereas, decision trees are easy to understand and can be applied easily.

There are three steps in data classification: (1) data preparation (data collection and data pre-processing), (2) construction of a machine learning model (training), and (3) using the machine learning model to classify unknown data samples (testing). We will describe Step 1 in Section 6.3 and Steps 2 and 3 will be detailed in the experiments in Section 6.4.

6.3. Characterization of Multi-threaded Applications

In this section, we describe the essential characteristics of multi-threaded applications in order to classify them with respect to their parallel patterns. We collect characteristics dynamically, that is, during the execution of the application, hence it is important to keep the characterization overhead low. Furthermore, several characteristics cannot be reliable obtained with a static analysis such as data sharing or thread communication.

Table 6.1. Characteristics of multi-threaded applications.

Characteristics	**Sub-characteristics**
General Threading	Program Counter (PC) Dynamic instruction count (IC) Creator thread Creation time Exit time Lifetime (LT)
Thread Communication	Ratio of Communicating Threads (RCT) Ratio of Communication Volume (RCV)
Data Sharing	Private Read-only Producer/Consumer Migratory

In Table 6.1, we show three major characteristics and their corresponding sub-characteristics. Note that we use the same characteristics that we described in Chapters 3 and 4. These characteristics are commonly used to describe the behaviors of threads for parallel pattern detection in the literature [30]. We use high level characteristics that capture the software architectural characteristics of an application. This is because parallel patterns are high level characteristics that define the structure of an application in terms of thread communication and data sharing behaviors. Hence, we cannot use low level microarchitecture dependent characteristics including CPI, cache miss rate, branch misprediction rate (that are typically obtained using hardware performance counters) since they capture performance characteristics instead of software architectural characteristics. For example, we cannot use hardware performance counters to collect data sharing information such as which thread reads/writes from/to which cacheline. Therefore, two applications with different parallel patterns can potentially have the same performance counter values and not allow us to detect the correct pattern.

Compared to the high number of characteristics used in the literature [39, 40], we chose a small number of characteristics. Having more characteristics such as synchronization constructs, instruction mix, and data flow can improve the accurate representation of each thread's behavior, but collecting and using a higher number of characteristics with machine learning techniques is costly as shown in the experiments.

We now give some examples of what each sub-characteristic captures and how it influences the classification of parallel patterns. While threads execute the same function (PC) with different inputs in geometric decomposition and recursive data parallel patterns, each thread executes a different function (PC) in pipeline and event-based coordination patterns. We have low RCV and RCT values and read-only and private data sharing in task parallel and divide and conquer patterns since there is no or few thread communications. On the other hand, we have high RCV and RCT values in geometric decomposition and recursive data patterns since threads communicate using shared memory with producer/consumer data sharing.

6.3.1. Data Preparation

6.3.1.1. Data Collection. The data in our case corresponds to the sub-characteristics of the given multi-threaded application. We use a DynamoRIO dynamic binary instrumentation tool and Umbra memory shadowing tool to collect these sub-characteristics during the execution of an application as described in Chapter 3. In [46], the experimental results show that the slowdown due to instrumentation varies from about 10% to 6×. The overhead (slowdown) comes from dynamic instrumentation system itself and the instrumentation clients, which are extra inserted instructions and application code modifications among others [44].

In addition, we use Pin tool [45] and develop a new client similar to our DynamoRIO client to collect the sub-characteristics of large applications. This is because our DynamoRIO client works on applications that have up to 32 threads. Hence, we use Pin in our data collection scalability experiments as we will show in Section 6.5.

In general, when using a dynamic binary instrumentation tool, collecting more characteristics takes proportionally more time. This is because collecting different characteristics requires instrumenting different parts of an executable code. Furthermore, there can be some characteristics that are more expensive to collect than others. For example, in our case, collecting RCV sub-characteristic is more expensive than collecting PC sub-characteristic since we need to monitor each read and write operations on every cacheline to gather RCV. Whereas, we only wrap thread creation operations to gather PC. Next, we describe the pre-processing of the collected data in order to construct models.

6.3.1.2. Data Pre-processing. Once the data (characteristics) is collected, we need to normalize the data in order to construct models. That is, we transform raw data into an understandable format. We now describe normalization for each group of sub-characteristics.

We normalize the general threading sub-characteristics by scaling each of them to 0 - 1 range. While 1 means that threads have the same sub-characteristics, 0 means that threads have different (unique) sub-characteristics. We use the mean M, the standard deviation S, and the coefficient of variation CV, which is defined as the ratio of the standard deviation S to the mean M, of a sub-characteristic in normalization. We normalize all general threading sub-characteristics except program counter and the creator thread as follows:

- If $CV \geq 0.2$, the normalized sub-characteristic $norm_x = 0$, for sub-characteristic x. Our training model determines this value of CV and 0.2 is the value that gives the most accurate models. Note that a higher CV value denotes inconsistency among the sub-characteristics within the group.
- Else the normalized sub-characteristic $norm_x = \#st \ / \ \#wt$, where $\#st$ is the number of threads that do not have the sub-characteristic x in the $2S$ range around M and $\#wt$ is the number of all worker threads.

For example, when we have a multi-threaded application with 4 worker threads, each having 2642, 2650, 2703, and 2705 dynamic instruction counts, respectively, M is 2675, S is 34, CV is 0.01, and the normalized dynamic instruction count sub-characteristic $norm_{IC}$ is 1 since $E = \#wt = 4$. The normalized sub-characteristic indicates that the threads are the same and balanced.

For normalizing the program counter sub-characteristic, we calculate the number of unique start (entry point) program counters (U) for all worker threads. Note that U ranges from 1 to $\#wt$. We then normalize the program counter as $norm_{PC} = (\#wt - U)/(\#wt - 1)$. Similarly, we can normalize the creator thread sub-characteristic. For example, if four worker threads run the same function, then U is 1, $\#wt$ is 4, and $norm_{PC}$ is 1. On the other hand, if each thread runs a different function, $U = \#wt = 4$, and $norm_{PC}$ is 0.

Our thread communication sub-characteristics are already normalized and are given as RCT and RCV described above. Note that different (raw) sub-characteristics that are gathered when using different inputs can result in the same normalized sub-characteristics, which is what we expect from normalization. For instance, when the total number of cachelines used and the number of cachelines used in communication are 100 and 10 in one case and 1000 and 100 in another case, the normalized RCV is 0.1 in both cases. We further address this issue in Section 6.5.

Finally, we calculate the normalized data sharing sub-characteristics as follows. For private data sharing sub-characteristics, we have $norm_{private}$ = *(number of private data sharing sub-characteristic / total number of all data sharing sub-characteristics).* We calculate other normalized data sharing sub-characteristics similarly. The next two steps in classification with machine learning techniques will be described in the next section.

6.4. Experiments

We performed experiments to validate our parallel pattern classification techniques. Our multicore machine uses an Intel i7 processor with 4 cores, 6MB cache, and 6GB of memory. Note that our techniques target CPU applications but similar techniques can be applied other architectures such as GPU. We classified 26 benchmarks from PARSEC [3] and Rodinia (OpenMP) [4] suites based on their parallel patterns. That is, our test set includes 26 benchmarks. Since our aim is to validate our techniques on these well-known and widely used benchmarks, we do not use them during training. We used simmedium inputs for PARSEC and default inputs for Rodinia. Note that although our framework is capable of running with larger inputs, we do not use them in our experiments because dynamic binary instrumentation can increase the execution time of an application drastically. Since PARSEC and Rodinia benchmarks only use task parallel, geometric decomposition, and pipeline among six possible parallel patterns, which we will experimentally show later as well, we also classified benchmarks including Sudoku, Fibonacci, and CarWashSim [30] that exhibit the remaining three parallel patterns.

For training purposes, we use 40 multi-threaded applications, which are either developed by us or used from other benchmark suites. Note that these are different from the above mentioned 26 benchmarks that will be used for testing purposes. In particular, for training, we use 7 benchmarks from EEMBC MultiBench [6] benchmark suite, which contains applications that help select the best hardware for embedded systems, and 9 benchmarks from open source benchmark suites including matrix multiplication, quick sort, and graph search. We cannot use each benchmark in the training set with multiple inputs to get a larger training set because when the mentioned training benchmarks are used with different inputs they result in the same characteristics and same parallel patterns as described in Chapter 3, hence not necessarily enriching our training set. We further address the usage of different inputs in Section 6.5. The remaining 24 benchmarks in the training set were developed by us according to the parallel pattern definitions in [30], where we developed 4 synthetic benchmarks for each of the six parallel patterns. Note that we developed synthetic benchmarks for the

training set because existing benchmarks do not cover all parallel pattern types such as event-based coordination. Ultimately, we created a high quality, balanced training set having diverse characteristics and covering the modeled (parallel pattern) space with the help of synthetic benchmarks. Our synthetics are such that the number of threads ranges from 2 to 24, the running times are between 0.01 seconds and 60 seconds, and the lines of code are between 50 and 1000. In summary, our training set is a 40×12 training matrix, where for each benchmark we have 12 normalized sub-characteristics. Since the size of our (high quality) training set is small, we also use k-fold cross validation to measure the accuracies of our techniques as described in Section 6.5. We ran each benchmark 10 times in order to obtain thread characteristics.

To implement machine learning techniques, we use MATLAB [53]. To collect characteristics of multi-threaded applications, we use DynamoRIO and Umbra as described in Section 6.3.1.1. Also, we use Pin as described in Section 6.3.1.1 in data collection scalability experiments as we will show in Section 6.5. At the end of the data preparation step described in Section 6.3.1, we obtain 12 normalized sub-characteristics: 6 general threading, 2 thread communication, and 4 data sharing sub-characteristics.

6.4.1. Pattern Classification Results

Table 6.2 shows our parallel pattern classification results on 26 benchmarks, none of which are used in training. The column *#wt* shows the total number of worker threads that are created during the execution of each benchmark. Note that the benchmarks we used cover a wide range of total number of worker threads. The column *Pattern* shows the parallel pattern of the benchmark found by us through code analysis. We also validated the parallel patterns of PARSEC benchmarks from the literature since they are available but Rodinia patterns are not available from the literature. Rodinia benchmarks use OpenMP *parallel for* construct [43] and if the data used in OpenMP is private (e.g. *reduction* construct), then it results in task parallel pattern, otherwise it results in geometric decomposition pattern. The columns *kNN*, *DT*, *NBC*, *NNW*, and *PCA* show the parallel pattern classification results of k-nearest neighbor, decision tree, naive Bayes classifier, neural network, and principal component analysis,

Table 6.2. Pattern classification results.

Suite	Benchmark	LOC	#wt	Pattern	kNN	DT	NBC	NNW	PCA
PARSEC	Blackscholes	1262	8	TP	TP	TP	TP	TP	TP
	Bodytrack	7696	9	GD	GD	GD	GD	GD	GD
	Canneal	2794	4	TP	TP	TP	TP	GD*	TP
	Dedup	7125	8	Pl	Pl	Pl	Pl	Pl	Pl
	Facesim	20275	5	TP	TP	TP	TP	TP	TP
	Ferret	10765	18	Pl	Pl	Pl	Pl	Pl	Pl
	Fluidanimate	2784	4	GD	GD	GD	GD	TP*	TP*
	Swaptions	1095	4	TP	TP	TP	TP	TP	TP
	X264	38546	15	Pl	Pl	Pl	Pl	GD*	GD*
Rodinia	Kmeans	2146	3	TP	TP	TP	TP	TP	TP
	HotSpot	196	3	GD	GD	GD	GD	TP*	EbC*
	Back Propagation	478	7	TP	TP	GD*	TP	TP	TP
	SRAD	495	1	TP	TP	TP	TP	TP	TP
	Breadth-First Search	125	3	TP	TP	TP	TP	TP	TP
	CFD Solver	1539	7	TP	TP	GD*	TP	TP	RD*
	LU Decomposition	541	3	GD	GD	GD	GD	GD	GD
	Heart Wall Tracking	2244	3	TP	TP	TP	TP	TP	TP
	Particle Filter	398	7	GD	GD	GD	GD	TP*	TP*
	PathFinder	127	3	GD	GD	GD	GD	GD	GD
	LavaMD	353	3	GD	GD	GD	GD	GD	EbC*
Other	SSCA2v2.2	23950	8	RD	RD	RD	Pl*	GD*	RD
	Sudoku	196	11	RD	RD	RD	RD	RD	RD
	Fibonacci	55	8	DaC	DaC	DaC	DaC	DaC	TP*
	Merge Sort	280	16	DaC	DaC	DaC	DaC	DaC	DaC
	CarWashSim	253	6	EbC	EbC	EbC	EbC	Pl*	RD*
	MultiTellerBankSim	416	11	EbC	EbC	EbC	EbC	Pl*	RD*

respectively. In the table, wrong classifications of parallel patterns are denoted by a '*' symbol. For example, `Back Propagation` is task parallel but it is classified as geometric decomposition (GD*) by our decision tree. Table 6.2 also shows that PARSEC and Rodinia benchmarks do not contain all parallel pattern types such as divide and conquer pattern, recursive data, and event-based coordination.

We will describe the details of experiments for each machine learning technique in the following sections. In the case of kNN technique the details can be found in Chapter 3, and we enhanced that work with feature selection in this chapter and discuss those specific results here.

We examine each machine learning technique in terms of accuracy, characterization overhead, speed, and memory consumption. The *accuracy* of a machine learning technique is the percentage of instances in the test set that are correctly classified. The accuracy is important because a wrong classification of a multi-threaded application results in the wrong description of its behaviors. We denote *accuracy* as *high* ($accuracy \geq 90\%$), *medium* ($90\% > accuracy \geq 80\%$), and *low* ($accuracy < 80\%$). The *characterization overhead* is the ratio of the running time of a multi-threaded application that is instrumented using DynamoRIO dynamic binary instrumentation and Umbra memory shadowing tools to the uninstrumented running time of the same application. We denote the average *characterization overhead* as *high* ($overhead \geq 10\times$), *medium* ($10\times > overhead \geq 5\times$), and *low* ($overhead < 5\times$). The *speed* is the amount of time needed for a machine learning to perform learning and classification that are done in Steps 2 and 3 of data classification. We denote the average *speed* as *high* ($speed < 0.03\,seconds$), *medium* ($0.1\,seconds > speed \geq 0.03\,seconds$), and *low* ($speed \geq 0.1\,seconds$). Similarly, *memory usage* is the amount of memory needed in Steps 2 and 3 of data classification and we denote the average *memory usage* as *high* ($usage \geq 10KB$) and *low* otherwise.

6.4.2. Decision Trees

6.4.2.1. Construction of a Decision Tree. We trained our decision tree using the Iterative Dichotomiser 3 (ID3) algorithm [13, 14] and obtained the decision tree shown in Figure 6.1. While building our decision tree, at each step, we decide the most important sub-characteristic and a test on this sub-characteristic. The most important sub-characteristic is the one which gives the largest split over the training set. We quantify the largest split with information gain as defined in [13, 14]. Information gain is the reduction in entropy, which characterizes the purity of a subset of the train-

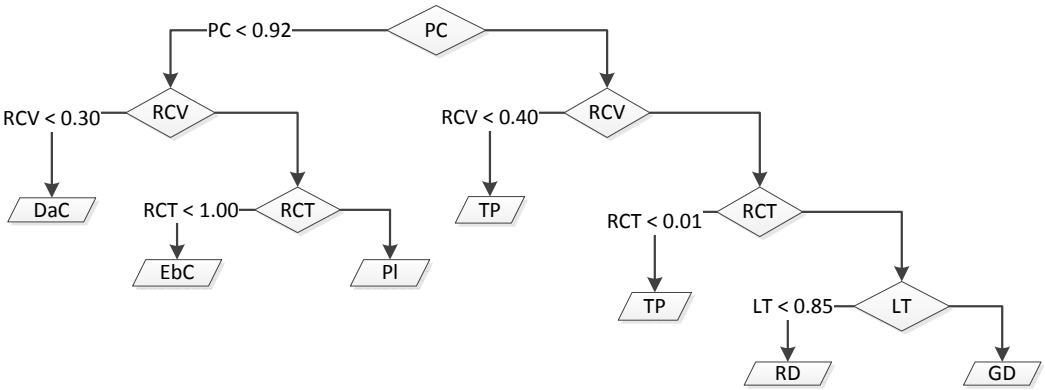

Figure 6.1. Decision tree for parallel pattern classification.

ing set, caused by splitting the training set according to a selected sub-characteristic. Note that the sub-characteristics we used are continuous variables based on which we selected an optimal point to yield the highest information gain. That is, we create an optimal point and then split the sub-characteristic list into those whose value is below the optimal point and those that are greater than or equal to it. We then create an internal node from the selected sub-characteristic and add one branch for each test leading out from this node. For example, in our decision tree, the first selected sub-characteristic is PC and the test is $PC < 0.92$. If all benchmarks on a test branch have the same parallel pattern, then we add a leaf node and assign the parallel pattern label. Otherwise, we continue creating new internal nodes and new tests until all benchmarks have the same parallel pattern on a branch. After producing the decision tree, we prune away internal nodes with low information gain to prevent overfitting.

The decision tree shows that we can classify the parallel pattern of a multi-threaded application by only using PC, LT, RCV and RCT sub-characteristics. This is because there exist implicit relations between sub-characteristics and we do not need to use all sub-characteristics to classify a multi-threaded application. For example, when threads have the same program counter, they generally have similar creation and exit times. Similarly, when we have high RCV or RCT values, we have high producer/consumer or migratory data sharing. Since we use only four sub-characteristics for classification, benchmark characterization can be sped up and four instead of twelve sub-characteristics can be collected.

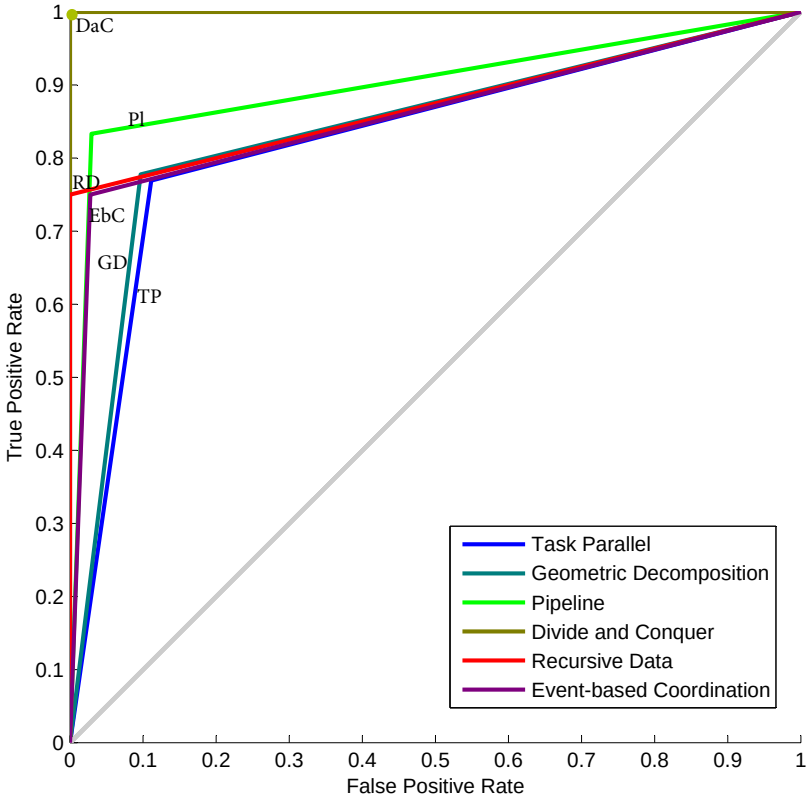

Figure 6.2. ROC curve of our decision tree.

Our decision tree correctly classifies 32 of the 40 multi-threaded applications from our training set with an 80% accuracy. In Figure 6.2, we show the performance of our decision tree for each parallel pattern as a Receiver Operating Characteristic (ROC) curve. Table 6.3 shows the maximum True Positive Rate (TPR) and maximum False Positive Rate (FPR) cut-offs of our ROC curve. The ROC curve shows that divide and conquer and recursive data parallel patterns have the lowest false positive rate (0.0) and divide and conquer parallel pattern has the highest true positive rate (1.0).

6.4.2.2. Using the Decision Tree. We show the parallel pattern classification results of our decision tree in column *DT* of Table 6.2. Our decision tree correctly classifies 24 of the 26 benchmarks in the test set with a 92.3% accuracy. Hence, the accuracy of our decision tree is high. The characterization overhead is 3.5× and low, since we use only four sub-characteristics for classification instead of twelve. The speed is 0.01 seconds and high, and the memory usage is 2KB and low.

Table 6.3. Cut-offs of our ROC curve.

Parallel Pattern	Maximum TPR cut-off	Maximum FPR cut-off
Task Parallel	0.77	0.11
Geometric Decomposition	0.78	0.10
Pipeline	0.83	0.03
Divide and Conquer	1	0
Recursive Data	0.75	0
Event-based Coordination	0.75	0.03

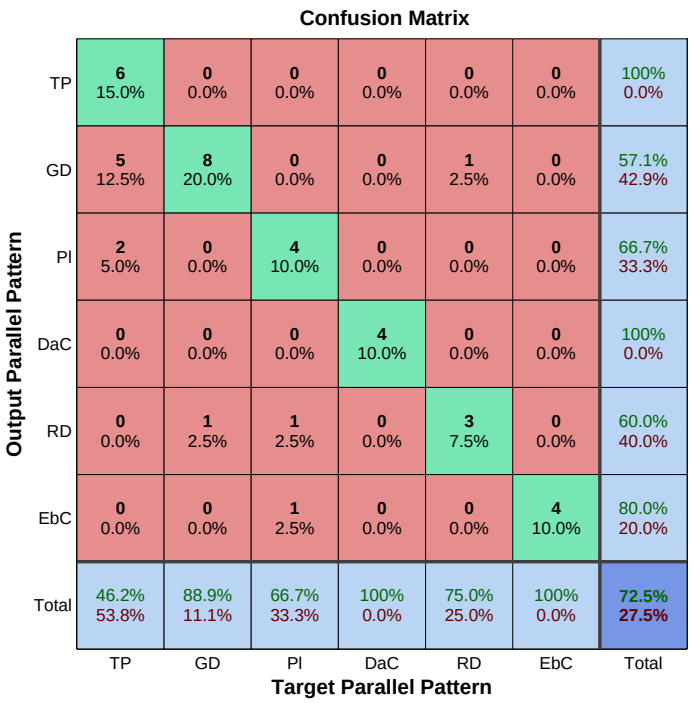

Figure 6.3. Confusion matrix of the naive Bayes classifier using the Gaussian method on the training set.

6.4.3. Naive Bayes Classifier

6.4.3.1. Construction of a Naive Bayes Classifier. For constructing our naive Bayes classifier using the Gaussian method, called NBC1, we use all sub-characteristics of 40 benchmarks in the training set. We trained the NBC1 using the algorithm in [13, 14]. Our classifier correctly classifies 29 of the 40 multi-threaded applications from our training set with a 72.5% accuracy. Figure 6.3 shows the confusion matrix of NBC1 on

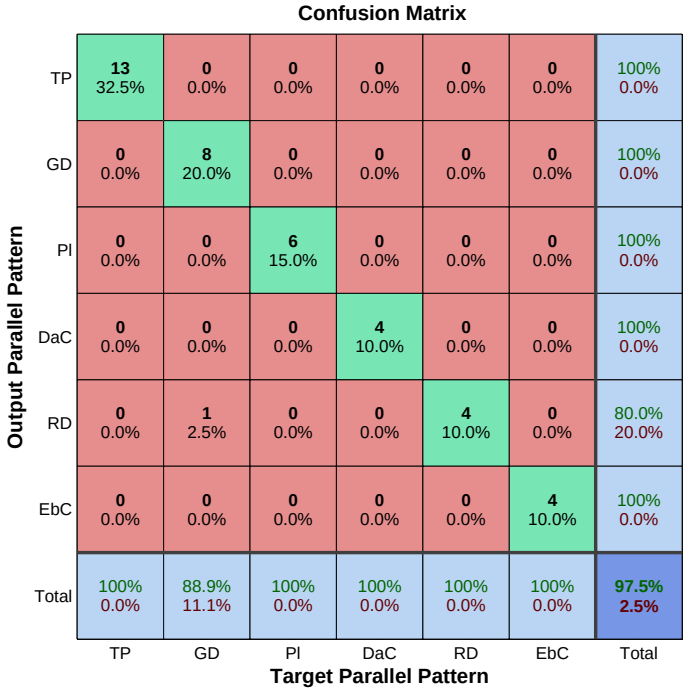

Figure 6.4. Confusion matrix of the naive Bayes classifier using the Kernel Density Estimation method on the training set.

the training set. In the figure, each column shows a target parallel pattern and each row shows the parallel pattern that is predicted by NBC1. The diagonal cells of the matrix show the number of multi-threaded applications that are correctly classified, and the off-diagonal cells show the number of the misclassified multi-threaded applications. For example, the cell at row 3 and column 1 shows that two task parallel applications are classified as pipeline. The cell in the bottom right shows the total percent of correctly classified multi-threaded applications (upside), which is 72.5% and the total percent of misclassified multi-threaded applications (downside), which is 27.5%.

We observe that the accuracy of NBC1 is low because it uses the Gaussian probability distribution method to model continuous values. To improve the accuracy, we use the Kernel Density Estimation (KDE) method [69] instead of the Gaussian method. Our classifier using the KDE method, which is called NBC2, correctly classifies 39 of the 40 multi-threaded applications from our training set with a 97.5% accuracy. Figure 6.4 shows the confusion matrix of NBC2 on the training set. The figure shows that only one geometric decomposition application is misclassified as recursive data application.

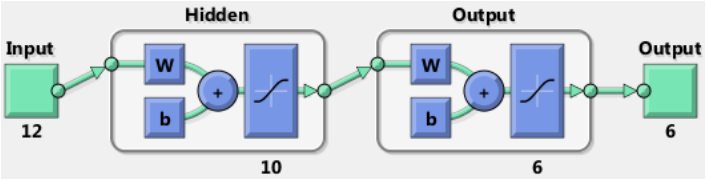

Figure 6.5. Configuration of our neural network.

6.4.3.2. Using the Naive Bayes Classifier. We show the parallel pattern classification results of our naive Bayes classifier NBC2 in column *NBC* of Table 6.2. NBC1 correctly classifies 19 of the 26 benchmarks with a 73.1% accuracy, whereas NBC2 correctly classifies 25 of the 26 benchmarks with a 96.2% accuracy. Hence, the accuracy of NBC2 is high. Since the accuracy of the NBC2 is higher than the NBC1, we refer to it as the naive Bayes classifier (NBC) from now on. The accuracy of NBC is high. The characterization overhead is 20.2× and high, since we use all twelve sub-characteristics for classification. The speed is 0.05 seconds and medium, and the memory usage is 3KB and low.

6.4.4. Neural Networks

6.4.4.1. Construction of a Neural Network. Our neural network is a two-layer feed-forward network, which has one hidden layer and one output layer as seen in Figure 6.5. In the training phase of our neural network, 12 sub-characteristics and the correct parallel pattern for each multi-threaded application in the training set are given. The size of the input layer is 12 since we have 12 sub-characteristics and the size of the output layer is 6 since we have 6 types of parallel patterns. Note that the output of our neural network is a vector that has 6 elements, where only one element is 1 and the others are 0. We set the size of the hidden layer to 10 in our experiments because we experimentally observed that this value shows better performance than other sizes such as 5 or 30.

We use scaled conjugate gradient method [70], which is a neural network training function that updates weight values, to train our neural network. We use Mean Square Error (MSE) as performance function, which is the average squared error between

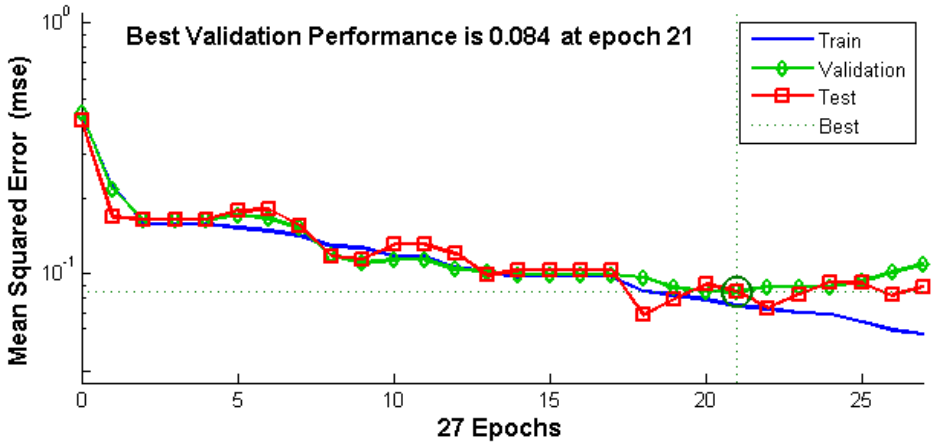

Figure 6.6. Training performance of the neural network.

the neural network outputs and the target parallel patterns. Our neural network has sigmoid transfer functions in the hidden layer and the output layer. We use 70% of the training set for training, 15% to validate that the neural network does not overfit, and 15% to independently test the neural network. A trained neural network can result in a different solution because of different initial weight values, different divisions of data into training, validation, and test sets. To overcome this problem, we retrain our neural network 10 times and select the one with the highest accuracy.

During the training, we use the magnitude of the gradient of performance and the number of validation checks to terminate the training. Note that validation checks represent the number of successive iterations that the performance fails to decrease. The training stops when the magnitude of the gradient is less than 1e-5 or the number of the validation checks reaches 6.

After configuring the parameters of our neural network as described above, we train our neural network. We show training, validation, and test performance in Figure 6.6. The training of our neural network stops at 27th iteration since the validation checks reach 6. We have the best validation performance, which is 0.084, at 21st iteration. Our neural network correctly classifies 27 of the 40 multi-threaded applications from the training set with a 67.5% accuracy and Figure 6.7 shows the confusion matrix of our neural network on the training set.

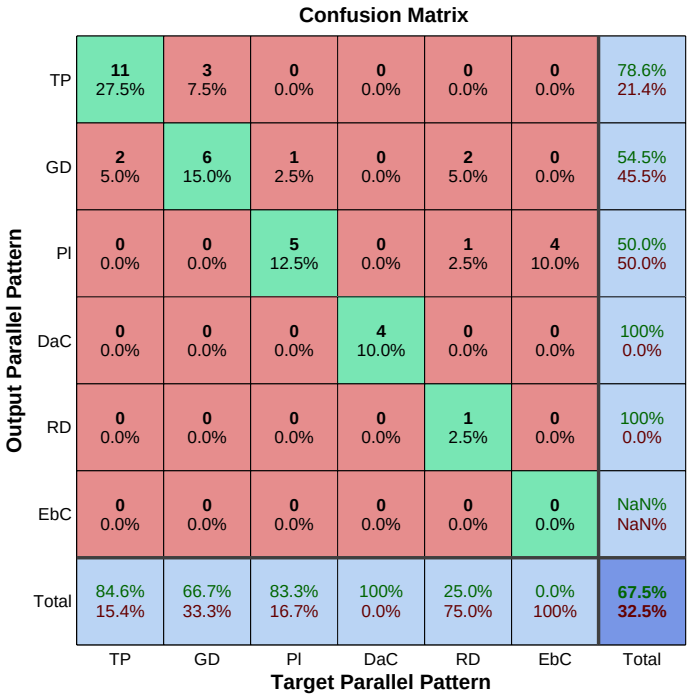

Figure 6.7. Confusion matrix of the neural network on the training set.

6.4.4.2. Using the Neural Network. We show the parallel pattern classification results of our neural network in column *NNW* of Table 6.2. Our neural network correctly classifies 18 of the 26 benchmarks in the test set with a 69.2% accuracy. Hence, the accuracy of our neural network is low. The characterization overhead is 20.2× and high, since we use all twelve sub-characteristics for classification. The speed is 0.31 seconds and low, and the memory usage is 6KB and low. Since NNW provides low accuracy, high characterization overhead, and low speed, it is not suitable for parallel pattern detection.

6.4.5. Principal Component Analysis with K-means

6.4.5.1. Construction of a Principal Component Analysis with K-means. In this section, we perform Principal Component Analysis (PCA) followed by k-means clustering on the training set. We use PCA to reduce the number of sub-characteristics that we use for parallel pattern classification while preserving significant variances of the raw sub-characteristics. We know that using a few number of features reduces the running time and memory usage of our clustering and classification algorithm. Figure

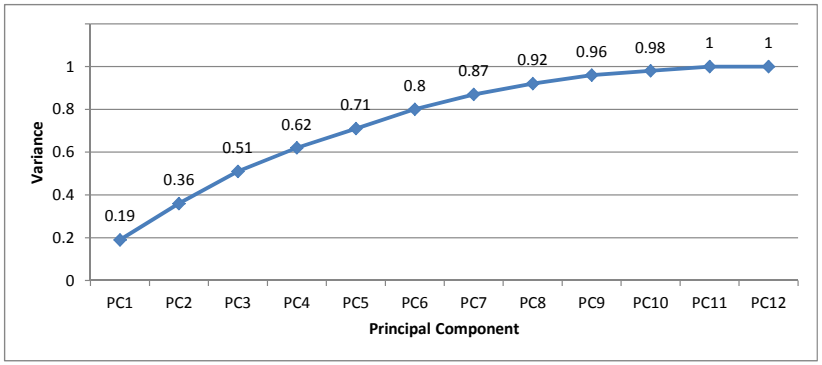

Figure 6.8. Principal components and their variance.

6.8 shows the application of PCA on the training set to find the principal components and their variances. Typically, most of the variance is contained in the first two or three principal components. However, in our case, we need to use the first six principal components that capture 80% of the total variance of the training set. Although the first nine principal components capture 96% of the total variance of the training set, using more number of principal components increases the CPU and memory usage of the clustering and classification techniques.

Once we find the principal components, we generate 6 clusters using k-means clustering algorithm [71]. Each cluster denotes a parallel pattern type and is represented by its centroid, which is a 6-dimensional vector. Since the clusters that k-means algorithm finds depend on the starting points, we run the k-means algorithm 10 times and select the solution that gives the minimum total sum of distances.

After finding the centroids of the clusters, we measure the accuracy of the PCA with k-means on the training set. We first find the principal components of the sub-characteristics of each application and then we measure the Euclidean distance between the principal components and each centroid. The parallel pattern type of the nearest centroid gives the parallel pattern of the application. PCA with k-means correctly classifies 32 of 40 applications in the training set with an 80% accuracy.

6.4.5.2. Using the Principal Component Analysis with K-means. We show the parallel pattern classification results of PCA with k-means in column *PCA* of Table 6.2.

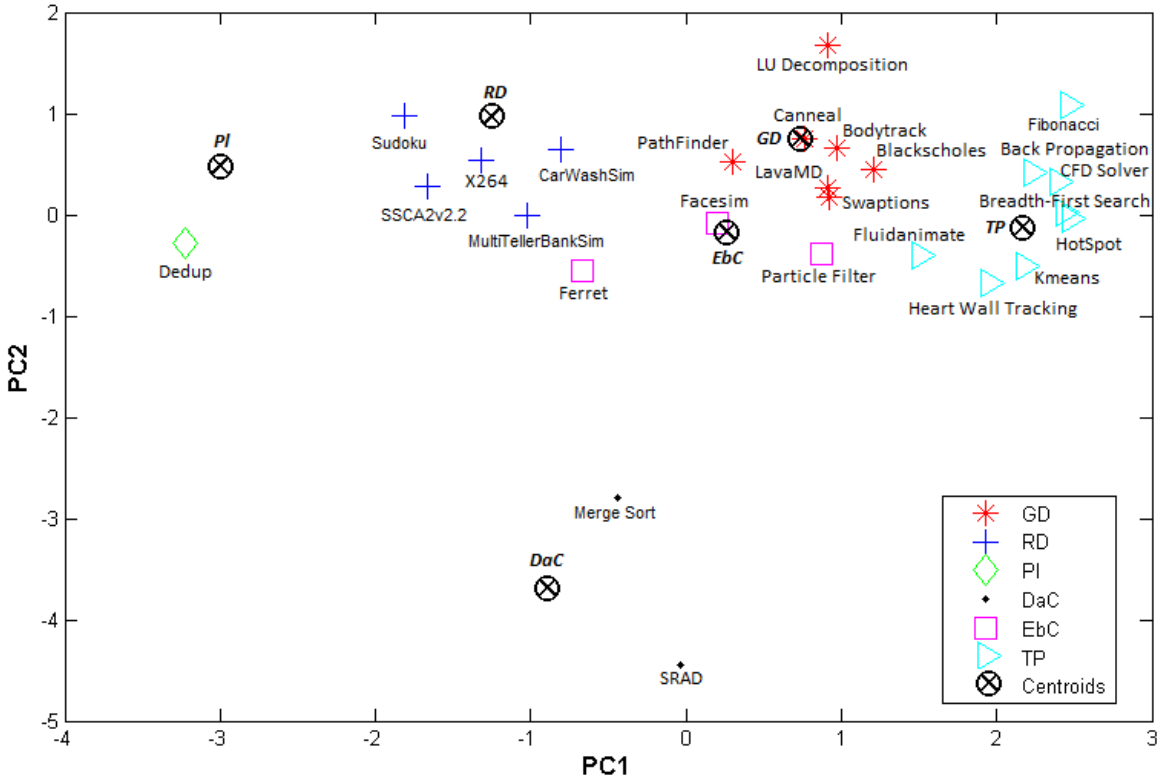

Figure 6.9. The PCA space that is projected into 2 dimensions (PC1 and PC2).

PCA correctly classifies 17 of the 26 benchmarks in the test set with a 65.4% accuracy.

We also performed a set of experiments to see whether using only the first two principal components can accurately classify the benchmarks. Note that using only two principal components improves the performance of the clustering and classification. Figure 6.9 shows the PCA space that is projected into 2 dimensions (PC1 and PC2) and the parallel pattern classification results of the 2 dimensional PCA with k-means. In the figure, we show six cluster centroid locations that represent parallel patterns and the parallel patterns of the benchmarks. The two dimensional PCA with k-means correctly classifies 13 of the 26 the benchmarks with a 50% accuracy. This is because the first two principal components only preserve 36% of the total variance of the training set. Although using only the first two principal components is fast and memory efficient, the accuracy is low. Our experiments show that using more than six principal components increases the accuracy up to 85% but decreases the speed and using less than six principal components decreases the accuracy but increases the speed. Hence, using the first six principal components is the optimal point where there is a balance between

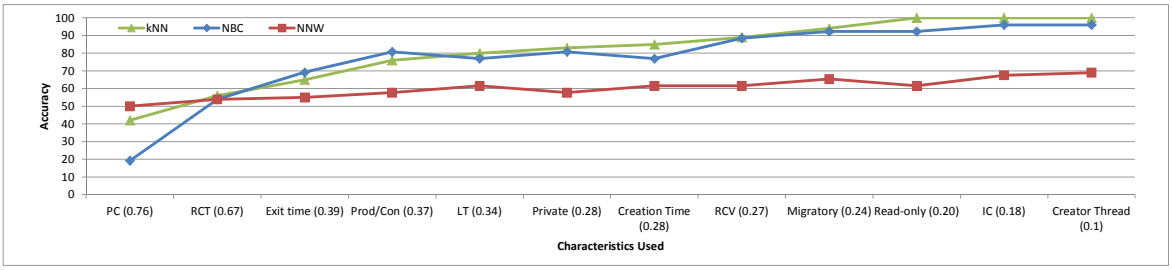

Figure 6.10. Accuracies of our machine learning techniques using the selected sub-characteristics proposed by MI.

Table 6.4. Comparison of machine learning techniques.

Metric	kNN	DT	NBC	NNW	PCA
Accuracy	High (100%)	High (92.3%)	High (96.2%)	Low (69.2%)	Low (65.4%)
Characterization Overhead without FS / with FS	High (20.2×/17.1×)	Low (3.5×/-)	High (20.2×/20.1×)	High (20.2×/20.2×)	High (20.2×/-)
Speed	Medium (0.04s)	High (0.01s)	Medium (0.05s)	Low (0.31s)	Low (0.53s)
Memory Usage	Low (2KB)	Low (2KB)	Low (3KB)	Low (6KB)	Low (9KB)

the accuracy and the speed. At this optimal point, the accuracy of our PCA with k-means is low. The characterization overhead is 20.2× and high, since we use all twelve sub-characteristics for classification. The speed is 0.53 seconds and low, and the memory usage is 9KB and low. Since PCA with k-means provides low accuracy, high characterization overhead, and low speed, it is not suitable for parallel pattern detection.

6.4.6. Feature Selection for Our Machine Learning Techniques

Since using a large number of features (characteristics) increases characterization, training, and learning costs, we decided to select a subset of relevant features for use in our model construction. For this purpose, we use the mutual information (MI) criterion [72]. MI measures the amount of information that the presence of a characteristic contributes to making the correct classification decision on a class. We ran Mutual Information Feature Selection (FS) on the training set and obtained MI for every feature ranked from the highest MI to the lowest. We then measured the accuracies

of our machine learning techniques using the sub-characteristics proposed by the MI and obtained Figure 6.10. Note that, in this chapter, we improve kNN that is used in Chapter 3 and [2] with feature selection where their pattern detection accuracies are 100% and 50%, respectively. Also, Table 6.4 shows the comparison of the machine learning techniques we used for parallel pattern classification. In the table, we present the characterization overheads without feature selection and with feature selection, if this applies to the particular technique, if not we denote it by a '-' symbol. Note that we do not apply FS on decision tree and PCA since our decision tree uses only 4 features and PCA is a feature reduction technique itself. From Figure 6.10 and Table 6.4, we observe that although the accuracy of kNN, NBC, and NNW do not improve using FS, the number of sub-characteristics required to achieve the maximum accuracies for kNN and NBC are slightly reduced. That is, instead of 12 sub-characteristics we can now use 10 and 11 sub-characteristics to achieve 100% and 96.2% accuracy with slightly reduced 17.1× and 20.1× average characterization overheads for kNN and NBC, respectively. Note that the FS provides 1.2× speedup for the kNN technique going from 20.2× to 17.1×. These results show that almost all the characteristics are relevant to the target class.

After applying FS, we analyze the characterization overheads of our techniques compared to the decision tree. Since our decision tree has the lowest average characterization overhead (3.5×), we find the characterization overheads of the other techniques where they have an accuracy that is equal or larger than the accuracy of our decision tree (92.3%). Our kNN has a 94% accuracy with a 15.1× average characterization overhead when we use the first 9 sub-characteristics proposed by the FS. Our naive Bayes classifier has a 96% accuracy with a 20.1× average characterization overhead when we use the first 11 sub-characteristics proposed by the FS. Our neural network cannot achieve a 92.3% accuracy. Although the FS reduces the characterization overheads for kNN and naive Bayes classifier, their characterization overheads are higher than the characterization overhead of our decision tree.

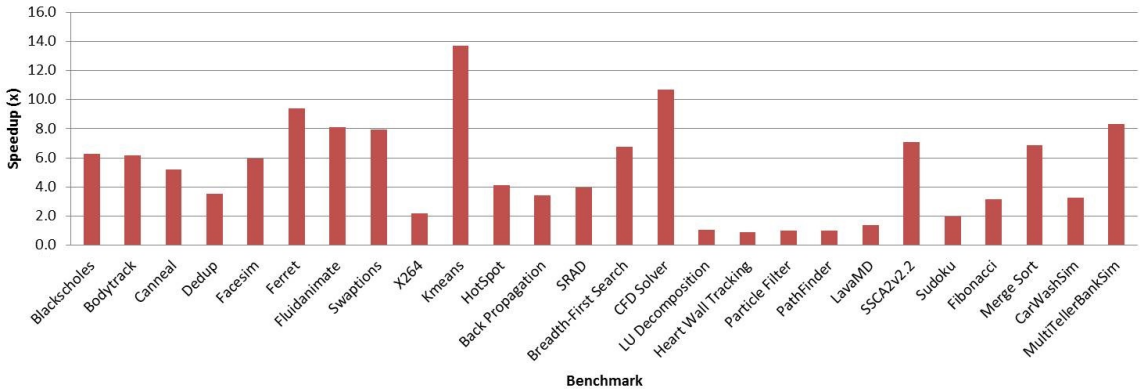

Figure 6.11. Characterization speedup of decision tree over kNN in Chapter 3.

6.4.7. Comparison of Our Machine Learning Techniques

We demonstrate the comparison of the machine learning techniques we used in Table 6.4. The characterization phase consumes more time than the training and classification phases. For instance, for `Bodytrack` benchmark, the characterization phase takes 51 seconds, whereas the training and classification phases using NBC take only 0.05 seconds. Hence, a technique with a low characterization overhead detects the parallel pattern of an application faster than a technique with a high speed.

Decision tree performs best in the characterization phase. Its overhead (3.5×) is lower than the other techniques since we collect only 4 sub-characteristics for the decision tree and 12 sub-characteristics for other techniques. Figure 6.11 shows the characterization speedup of decision tree over kNN technique without feature selection for each benchmark. Note that we calculate the speedup as the ratio of the characterization overhead for kNN to the characterization overhead for the decision tree. From the figure, the maximum speedup is 13.7× for `Kmeans` and the minimum speedup is 1.1× for `Heart Wall Tracking`. The decision tree has higher speedups for the benchmarks that are long running and memory intensive such as `CFD Solver` and `Ferret`. The average characterization overhead for kNN is 20.2× without feature selection and 17.1× with feature selection, hence the average characterization speedup of decision tree over kNN is 5.7× and 4.8×, respectively. We observe similar speedups for decision tree over other techniques as well, since they also use 12 sub-characteristics.

Decision tree also performs best in the training and classification phases as it is the fastest (0.01 seconds), uses less memory (2KB) than other techniques.

We observed that the neural network and PCA with k-means are not suitable for parallel pattern classification because of their low accuracies and low efficiencies. The accuracy of kNN, NBC, and decision trees are high, and are 100%, 96%, 92%, respectively. Hence, when there is a need for accurate classification, one can use the kNN, NBC, or decision trees. On the other hand, when there is also need for high speed one can use the decision tree.

6.4.8. Using Parallel Patterns Detection Results

6.4.8.1. Synthetic Benchmark Generation. We performed experiments to validate the usefulness of our parallel pattern detection techniques by adding our techniques to the MINIME tool that we developed. Details of this approach can be found in Chapter 3 where MINIME uses parallel patterns in capturing important characteristics of multi-threaded applications and generates synthetic multicore benchmarks from those applications.

In the experiments, we generated multi-threaded synthetic benchmarks from PARSEC [3] and Rodinia (OpenMP) [4] benchmarks. Our synthetics use the Pthreads library for multi-threading. We set the overall similarity score to 90% and individual similarity scores to 80%. Table 6.5 shows our pattern detection and synthesis results that we previously obtained in Chapter 3. Note that we used kNN technique in the experiments since it is the most accurate pattern recognition technique and we know that accuracy is more important than efficiency for synthetic benchmark generation. We show the lines of code and parallel patterns of original benchmarks (that is found from the literature) as *LOC* and *Parallel Pattern* in the table, respectively. In the table, we also show the number of iterations it takes to generate the synthetic benchmark (*#iter*), the speedup obtained using the synthetic in terms of execution time *Speedup(×)*, and the reduction in lines of code going from the original to the synthetic *CodeSize(×)*.

Table 6.5. Pattern recognition and synthesis results.

Original Benchmark				Synthetic Benchmark			
Suite	**Benchmark**	**LOC**	**Parallel Pattern**	**#iter**	**Speedup(×)**	**CodeSize(×)**	**OSS**
PARSEC	Blackscholes	1262	TP	2	10	10	95
	Bodytrack	7696	GD	16	11	19	90
	Canneal	2794	TP	2	22	20	93
	Dedup	7125	Pl	9	36	16	94
	Facesim	20275	TP	5	15	159	94
	Ferret	10765	Pl	5	67	7	90
	Fluidanimate	2784	GD	2	15	8	90
	Swaptions	1095	TP	2	13	7	91
	X264	38546	Pl	12	26	41	92
Rodinia	Kmeans	2146	TP	15	36	11	90
	HotSpot	196	GD	10	16	1	94
	Back Propagation	478	TP	5	20	3	94
	SRAD	495	TP	17	12	2	93
	Breadth-First Search	125	TP	18	35	0	94
	CFD Solver	1539	TP	11	10174	6	90
	LU Decomposition	541	GD	32	10	2	94
	Heart Wall Tracking	2244	TP	16	34	12	90
	Particle Filter	398	GD	9	10	1	91
	PathFinder	127	GD	14	13	0	95
	LavaMD	353	GD	19	30	1	90

For most benchmarks, we generate the synthetic benchmark in less than 20 iterations. In the table, speedup *Speedup(×)* and code size reduction *Code(×)* show that our synthetic benchmarks are much faster and smaller than original benchmarks. We denote *Code(×)* as 0 in cases where the original code size is very small and the corresponding synthetic code size is larger than the original. However, even in these cases we have faster synthetics which is our goal in generating synthetics. That is, the original benchmarks have larger and time consuming loops comparing to our synthetic benchmarks. It is also clear that we have a larger speedup or code size reduction for the benchmarks with high execution time or code size. In the table, we show the overall similarity score (*OSS*) where the average is 92%. These scores demonstrate that synthetics are above the target set by the user (90%) and have high degree of similarity with the originals.

6.4.8.2. Correlation of Parallel Pattern with Synthetic Benchmark Similarity. We validate the usefulness of parallel pattern detection results by analyzing the correlation between the parallel pattern score and the overall similarity score as briefly discussed in Chapter 3. Note that when using kNN, the parallel pattern scores are assigned to be inversely proportional to the distances to the application characteristics, where the highest score is 100 and the lowest score is 0.

To check this correlation, we synthesis six different synthetic benchmarks for each original benchmark where each has a different parallel pattern. Note that only one of them has the correct pattern. We obtain these synthetic benchmarks only after one iteration step because our goal is not to generate accurate synthetic benchmarks, it is to assess the usefulness of parallel patterns. After generating synthetic benchmarks, we calculate the parallel pattern score and overall similarity score for each synthetic. By analyzing the calculated six parallel pattern scores and overall similarity scores, we observed that we have the highest overall similarity score for the synthetic benchmark that has the highest pattern score, that is, the synthetic with the correct parallel pattern. For example, when `Bodytrack` (or `Kmeans`) has the correct parallel pattern, the overall similarity score is 61 (70), whereas the score is between 24 and 30 (32 and 53) for other parallel patterns. Note that the overall similarity scores in the correlation experiments are low since we generate the synthetics in only one iteration and we do not add code blocks to improve similarity scores. This is because our aim is to measure the influence of correct pattern detection and adding code blocks for improving similarity scores hides this influence.

Our experimental results also show that there is a linear correlation between the pattern score and overall similarity score. The correlation coefficients are 0.96 and 0.86 for `Bodytrack` and `Kmeans`, respectively. The average correlation coefficient is 0.71 for all benchmarks used during usefulness validation experiments.

Next, we perform the above correlation experiments to all of our pattern recognition techniques. We again generate synthetic benchmarks in one iteration by using the parallel pattern detected by the parallel pattern detection technique. We observed that

overall similarity scores are 59, 55, 57, 47, and 44 for kNN, decision trees, NBC, neural networks, and PCA with K-means, respectively. These scores validate the usefulness of our parallel pattern classification results where the most accurate techniques (kNN, decision trees, and NBC) result in the most accurate synthetic benchmarks. Furthermore, the synthetic benchmarks are similar on average 92% to the original benchmarks, the average speedup is 23× (without CFD Solver) and the average code reduction is 20×.

6.5. Discussion

6.5.1. Data Collection Scalability

The experimental results show that our current set of characteristics ensures accurate and efficient parallel pattern detection. Although adding more characteristics including synchronization and instruction mix characteristics can further improve accuracy or efficiency, this increases the characterization overhead. As characterization (data collection step of data preparation) is the most time consuming step, we prefer not to add more characteristics.

Improving data collection overhead can improve the overall speed dramatically since the main overhead comes from data collection. One option for this is to employ static (code) analysis. If we can obtain the source code of the application then we can gather PC and creator thread sub-characteristics by static analysis. Note that we cannot gather the remaining characteristics by static analysis which all depend on the running environment and dynamic behaviors. We performed a simple experiment to check whether using only the characteristics gathered statically (PC, creator thread) can result in an accurate model. In this case, the accuracy of the kNN was only 17%. Also, in Chapter 3, we use kNN and observe that the combination of data sharing and thread communication characteristics has the highest correlation with the parallel pattern with a 0.91 correlation coefficient unlike the static general threading characteristics.

Another option to improve data collection overhead is to use hardware performance counters. However, as we described in Section 6.3, we cannot use performance counters to collect our characteristics. We validated the scalability of our data collection step by successfully characterizing large Mozilla Firefox and Chromium Web Browser applications using our binary instrumentation technique. Specifically, we observed that the number of threads in Firefox ranges from 49 to 70 depending on the operations performed on the browser. Similarly, Chromium has 24 threads in the default case. Once we collect the characteristics of these applications, we normalize them and detect their parallel patterns similar to the benchmarks we used.

The performances of testing and training steps are determined by the amount of normalized data, which does not change when the number of cores is increased. When the number of cores is increased, if the application's performance increases then similarly data collection performance increases. This is because dynamic binary instrumentation mainly inserts code into the original application code. Furthermore, DynamoRIO and Umbra tools are meant to be efficient and scalable [44, 46].

6.5.2. Training Set Size and Cross Validation

Machine learning techniques are sensitive to the size and quality of the training set. Hence, there is need for a large benchmark suite or the training set should be high quality if it is small. Since there are not many publicly available multi-threaded benchmarks suites, we end up with a small training set. To remedy this situation, we carefully select applications to obtain a high quality training set where it is composed of applications with different characteristics and parallel patterns as described in the experiments in Section 6.4.

Since the size of our training set is small, we also use k-fold cross validation to measure the accuracies of our techniques. In the experiments, we use 66 benchmarks (40 from our training set and 26 from our test set). The average accuracies are 80, 82, 50, and 48 for decision tree, naive Bayes classifier, neural network, and PCA with k-means, respectively. We do not apply k-fold to kNN technique because it assumes

that there exist six applications that represent the reference behaviors of each parallel pattern in its training set and upon using k-fold a representative from each parallel pattern type is not necessarily guaranteed in the test set and results in poor accuracy.

The average accuracies in k-fold validation are lower than the accuracies observed in our earlier experiments since the training and test sets in k-fold validation do not contain all characteristics and all parallel patterns in many cases. To explain this situation, we give the following example. In case where k-fold generates a training set with no application with pipeline parallel pattern and a test set where many applications have pipeline pattern, the accuracy is going to be poor. However, the relative order of average accuracies using k-fold for decision trees, naive Bayes classifier, neural network, and PCA with k-means (80, 82, 50, and 48) is similar to the ones observed in our experiments (92.3, 96.2, 69.2, and 65.4).

Another option to increase the size of the training set is to apply multiple inputs on the existing benchmarks, which we explain next.

6.5.3. Impact of Multiple Inputs

In order to analyze the impact of input changes on our techniques, we performed a set of experiments using our decision tree technique. In the experiments, we generated two different training sets (ts-small and ts-medium) using small and medium inputs that is available for the benchmarks in the training set. Note that our results are valid if other inputs are used as well. We then used a single test set for validation.

We now show that our normalization works correctly and for both input types the prediction outcome is the same. Furthermore, adding both inputs to the same training set and obtaining such a single set does not enrich the quality of our training set.

We observed that although application characteristics in ts-small and ts-medium are slightly different (as the behavior of an application may change when its inputs change), the parallel pattern is the same. We use the relative ratio of each sub-

characteristic in normalization, hence our technique correctly normalizes potentially different characteristics. That is, the normalized characteristics of applications in both ts-small and ts-medium have the same normalization values. For example, if data sharing sub-characteristics of a benchmark with small input are 10, 20, 10, and 60 (for private, read-only, producer/consumer, migratory) then they are typically 100, 200, 100, and 600 for medium input (scaled version of the small input). In both cases, the normalized sub-characteristics are the same and are 0.1, 0.2, 0.1, and 0.6.

Since we have the same normalized characteristics for both ts-small and ts-medium, the same efficiency and accuracy is achieved with decision trees for the training set as well as the test set. If we join and use ts-small and ts-medium as a single training set, we obtain duplicate elements and a low quality training set. Hence, we conclude that using multiple inputs to get a larger training set does not improve the effectiveness of our techniques.

6.5.4. Multiple Parallel Patterns

Although each benchmark that we used in this thesis utilize only a single parallel pattern, we note that an implementation can potentially have a composition or hierarchy of patterns. However, we are not aware of a publicly available multi-threaded benchmark suite that includes benchmarks with a mixture of multiple patterns. Hence, one would need to find/create such real life applications for performing experiments to validate the pattern detection techniques, which detect multiple patterns. Therefore, we do not consider such behavior in this thesis.

6.6. Summary

Detecting the parallel pattern used in a multi-threaded application is crucial to the development and optimization of both multicore hardware and software. However, this topic has not been widely studied. We present a novel approach to parallel pattern detection using several machine learning techniques. We use high level thread communication and data sharing characteristics to capture the software architectural patterns

of multi-threaded applications since low level characteristics such as IPC, cache miss rate, and branch misprediction rate can only capture performance characteristics.

Our experiments show that machine learning techniques can accurately and quickly detect the parallel patterns of the benchmarks from PARSEC and Rodinia suites. Specifically, k-nearest neighbor, naive Bayes classifier, and decision trees are the most accurate techniques with a 100%, 96%, and 92% accuracy, respectively. Decision trees provide a 5.7× average characterization speedup over the other techniques that do not use feature selection and a 4.8× speedup over the other techniques that use feature selection. Overall, the decision trees are the fastest technique with the lowest characterization overhead producing the best combination of detection results. We also observe that the neural network and PCA with k-means techniques are not suitable for parallel pattern detection because of their low accuracies and low efficiencies. We validated the usefulness of the proposed techniques on synthetic benchmark generation.

7. RELATED WORK

In this chapter, we discuss the related work that falls into three main categories. The first category is the generation of synthetic benchmarks for CPUs. The second category includes a novel benchmark generation technique for GPUs. The third category is the previous work on machine learning techniques to detect parallel patterns.

7.1. Synthetic Benchmark Generation for Multicore CPUs

7.1.1. Software Architectural Patterns

In [28], the authors introduced design patterns for object-oriented programming. In [27], the authors present architectural patterns for parallel multi-threaded applications. Goswami et al. [73] describe pattern-based approaches in parallel computing and present a set of architectural-skeletons. Aldinucci et. al [63] demonstrate how a set of re-usable building blocks that support efficient design of parallel skeletons/patterns, can be used to design general purpose programming model and domain specific patterns. In [74], the authors presented a novel method for representing skeletons, which are high-level representations for parallel applications, and their composition. In [75], the authors discuss strengths and weakness of data parallel patterns. Asanovic et. al [76] introduce dwarfs, which are used to design and develop parallel programming models and architectures. A dwarf defines an algorithmic method that captures computations and communication patterns of an application.

7.1.2. Benchmark Characterization

There has been prior work on characterizing benchmarks mentioned in this thesis. In [3], the authors analyze several characteristics such as data locality, effects of different cache block size, degree of parallelization, and temporal and spatial behavior of communication for PARSEC. Che at al. [4] present characteristics of Rodinia benchmarks based on inherent architectural characteristics, parallelization, synchronization,

communication overhead, and power consumption. A comparison of PARSEC and Rodinia benchmark suites is given in [77], using instruction mix, working set, and sharing behavior characteristics. Their work shows that many of the workloads in Rodinia and PARSEC capture different aspects of performance metrics and they complete each other. This thesis shows that PARSEC and Rodinia benchmarks do not cover all parallel patterns. Poovey et al. [78] characterized the EEMBC benchmarks on different ISAs using trace-driven simulation. They used microarchitecture independent characteristics such as dynamic instruction percentages (integer, floating-point, load, store, branch, e.g.) and microarchitecture dependent characteristics such as cache miss rate and branch misprediction ratios. They mainly focus on the analysis of the similarity of performance characteristics among benchmarks.

There has been several works to understand and optimize the performance of workloads by using performance counters [56–58]. These studies use microarchitecture dependent metrics like cycles per instruction, cache miss rate, and branch misprediction rate. These studies are helpful in understanding performance and finding bottlenecks. Hoste et al. [10] introduced microarchitecture independent characteristics, some of which we also use in this thesis to compare benchmarks. Hillenbrand et al. [79] present an architecture independent methodology for analyzing communication of multi-threaded applications. The communication patterns they use are read-only, read/write, producer/consumer and migratory. Bharathi et al. [80] characterized workflows from different scientific communities.

7.1.3. Synthetic Benchmark Generation

Several works use statistical simulation for synthetic workload generation [81–84]. However, in these works mainly single threaded applications and microarchitecture dependent characteristics have been used, whereas this thesis targets multi-threaded applications and uses microarchitecture independent characteristics.

So far, synthetic benchmarks have mainly been developed for sequential applications [9,39]. There are some recent synthetics for multi-threaded applications in [58,85],

and [40] that target performance and power characteristics and may have lower error scores. These synthetics do not target embedded multicore systems, which we can, thanks to using MCA libraries. Note that MCAPI which is a lightweight message passing API and MRAPI which specifies essential application-level resource management capabilities are two of the standards developed by MCA [20,38] for closely distributed embedded systems. Since these standards are new they are lacking benchmarks. Our benchmarks will also serve to proliferate the usage of these standards among potential users. Also, their synthetics use low level assembly language, whereas we generate multi-threaded benchmarks as readable and portable C code. Specially, their synthetic benchmarks are generated for a specific number of cores and they need to be regenerated when the target system has different number of cores. However, we do not generate synthetic benchmarks for a specific number of cores and multi-threaded synthetic benchmarks can be used to quantify performance differences for studies when the number of cores are changed. We do not use simulators, which may consume a long time, for gathering application characteristics, rather we use dynamic binary instrumentation and performance counters. The number of characteristics we use (13) is smaller than theirs (around 40). Finally and most importantly, we use an even higher level of characteristics, that is, the parallel pattern that captures inherent characteristics of a multi-threaded application.

In [86], the authors proposed an approach for benchmark generation from behavior of real applications that is captured as statistical execution profile constructed from hardware performance counters. They target ARM-based mobile devices whereas we target CPU devices. In [87], the author proposed a synthetic workload generation technique in which the characteristics of the original workload are obtained from the system functional or logical resource usage and a synthetic set of jobs are constructed that places similar demands on the system resources. Wang et al. [88] use performance cloning to emulate cache organizations on real hardware. They specifically aim to mimic cache behavior, however, we mimic IPC, cache and branch prediction behaviors as well as parallel patterns. In [89], the authors present an automatic workload extraction approach for embedded software performance estimation. They use an LLVM-based characterization to collect target performance characteristics from the

source code, whereas we use performance counters to capture performance characteristics. In [90], we present a framework that automatically generates system level synthetic benchmarks from traditional benchmarks.

Portability of synthetics with changing microarchitectures has been studied in the literature [10, 40, 91]. In [92] and [93], the authors use LLVM compiler to generate ISA independent portable benchmarks. In particular, they generate synthetics for P4080ds system and their average error in IPC is 37.9% with maximum error of 212%. In [94], the authors generate portable and human-readable communication benchmarks in C for MPI applications. Their synthetics preserve only the communication characteristics and the execution time of original applications. In this thesis, we generate portable and human-readable synthetics. We performed portability experiments with changing microarchitectures as well as ISAs. Our synthetics target Pthreads applications and they preserve several high level and low level characteristics of the originals, while being faster than the originals.

Jung et al. [95] proposed a random synthetic program generator that is used to train machine learning models by collecting low level characteristics from the source code of an application. Then they use these models to predict and modify the application and its characteristics by using the best data structures for a given input/architecture combination. This approach is orthogonal to our approach. In that, we instrument the binary of the application to collect both low level and high level characteristics including thread communication and data sharing to generate a fast synthetic benchmark that preserves the characteristics of the application.

7.2. Thread-level Synthetic Benchmark for CPUs

7.2.1. Benchmark Characterization

In [96], the authors propose ScarPhase, a performance counter based phase detection library, to detect and classify phases in serial and parallel applications. They collect per thread performance characteristics including cycles per instruction (CPI)

and L3 miss ratio where they use Linux perf to obtain these characteristics similar to us. They validate their technique on multi-threaded application from PARSEC suite and show that PARSEC benchmarks have a diverse set of phase behaviors. However, we show that PARSEC benchmarks do not contain all parallel patterns such as event-based coordination. Perelman et al. [97] enhanced the single threaded SimPoint [98] to detect phases in parallel applications. They show that CPI and cache hit rates are the key characteristics to understand the performance and phases of a multi-threaded application. Similarly, we use per thread IPC and cache miss rates as performance characteristics in benchmark synthesis. In [99], the authors characterize multi-threaded applications in order to analyze the effect of shared-resource contention on performance. They use several characteristics including L1, L2, and L3 cache misses and perform characterization of the multi-threaded PARSEC benchmarks on real hardware. In this thesis, we also use per thread cache misses to capture and mimic performance characteristics of multi-threaded applications.

7.3. Synthetic Benchmark Generation for GPUs

7.3.1. Benchmark Characterization

In [19], the authors present a set of microarchitecture independent GPU characteristics that capture important behaviors of GPU applications: kernel stress, kernel characteristics, divergence characteristics, instruction mix, and coalescing characteristics. They use these characteristics with PCA and hierarchical clustering analysis to analyze diversity of GPU benchmark suites. We also use many of these characteristics and apply PCA to validate the importance of our characteristics. Che et al. [15] identify a set of important GPU characteristics, which are similar to our characteristics, including instruction throughput, computation-to-memory access ratio, and memory instruction mix to predict the performance of GPU applications by correlating their characteristics to existing applications' characteristics. The authors also conduct a PCA to illustrate similarity among benchmarks. Bakhoda et al. [17] present performance characteristics and performance bottlenecks of CUDA applications by analyzing characteristics including IPC, instruction mix, memory coalescing, and warp occupancy

on different hardware configurations. Similarly, Kerr et. al [100] use different characteristics that are similar to ones we use to identify relationships between application behavior and performance on different heterogeneous systems. Since we preserve the characteristics of an original application across different platforms, one can use our synthetic benchmarks, which are fast and small, in performance characterization and bottleneck identification studies instead of using large original applications. In [101], the authors introduce the NUPAR benchmark suite including OpenCL and CUDA applications and they characterize these applications in terms of several characteristics including occupancy, register utilization, local/shared memory utilization which are similar the ones we use. Rodinia and Parboil benchmark suites [4, 5] target heterogeneous multicore systems. In [16], the authors characterize the performance of OpenCL benchmarks from NAS Parallel Benchmark suite.

7.3.2. Synthetic Benchmark Generation

Synthetic benchmarks for CPU applications have been widely developed such as in [40,102] that target performance and power characteristics. These synthetics do not target GPUs. There exist only a few recent works on synthetic benchmark generation for GPUs. In [50], the authors generate miniature (synthetic) proxies of CUDA General Purpose Computing on Graphical Processing Unit (GPGPU) kernels where they mimic performance characteristics. Also, they only focus on kernel programs and they do not mimic host programs and data passing between CPU and GPU. This is because our goal is to accelerate GPU architecture simulation. They obtain 49× average and 589× maximum speedup with an average IPC error of 4.7% where we obtain 541× average and 7284× maximum speedup with an average IPC error of 9% and overall error of 4%. Note that they measure the similarity (accuracy) between the original and synthetic only in terms of one characteristic (IPC). However, we use eleven characteristics in similarity measurement and validate that our synthetics preserve all of them. Their approach cannot generate faster synthetics for benchmarks that do not execute any loops, whereas our approach generated faster synthetics for all benchmarks in the experiments. They generate synthetics in CUDA and target NVIDIA GPUs and they

embed assembly code in CUDA. However, our portable synthetics are in OpenCL and do not include assembly code and we target different GPUs platforms including AMD, NVDIA, and Intel. We also validated our approach on a simulator and real hardware, whereas they only validated on a simulator.

Huang et al. [103] use sampling technique to speed up GPU architecture simulation for CUDA applications where they achieve up to 10× speedup whereas we achieve up to 7284× speedup. Similarly, Lee et al. [104] parallelize the GPU architecture simulation where they gain up to 4.15× speedup.

In [105], the authors present a code generator that produces matrix multiply kernels written in OpenCL from a set of user given parameters. However, they do not mimic the characteristics of existing applications and also they only target matrix multiplication applications whereas we can generate synthetic benchmarks for different types of applications. PARAPHRASE FastFlow [62] provides OpenCL-based heterogeneous skeletons that make easier to understand and develop GPU applications for hybrid CPU/GPU architectures. Similarly, our small and accurate synthetic benchmarks and collected application characteristics can be used to understand large existing applications. In [106], the authors present a directive-based API, Dymaxion++, to enable programmers to optimize memory access patterns. This approach is orthogonal to our approach. They obtain an average of 3.2× performance improvement where we obtain on average 541× speedup. They also need the source code of original applications since they use source-to-source code translation. However, we only need the binaries of original applications.

7.4. Machine Learning Techniques to Detect Parallel Patterns

7.4.1. Parallel Pattern Detection

Poovey et al. [2] detect parallel patterns using kNN technique (although they do not explicitly say kNN). Our characteristics are similar to theirs. They detect parallel patterns from PARSEC benchmarks with 50% accuracy. We also used kNN

to detect parallel patterns but with 100% accuracy using 13 sub-characteristics. In these works, the authors define a set of reference behaviors that capture the important characteristics that each parallel pattern exhibits. The Euclidean distance is measured between the collected characteristics of a multi-threaded application and characteristics of each reference behavior. The authors define k as 1 where the class of a data sample is assigned to the class that is most common among its k nearest neighbors. Hence, the parallel pattern of the multi-threaded application is classified as the parallel pattern of its single nearest neighbor in terms of reference behaviors.

We demonstrated ways to use several machine learning techniques as well as kNN in this thesis. Our techniques are highly accurate such as kNN, decision trees, and naive Bayes classifier. We improve the efficiency of kNN and other techniques with feature selection, which has not been used for kNN earlier. Furthermore, we show that decision trees are not only accurate but also are the fastest technique with the lowest characterization overhead as only 4 sub-characteristics are used.

Parallel patterns have been used by Poovey et al. [59] for dynamic thread mapping where they implement the same pattern recognition algorithm in [2] in hardware, since their algorithm needs to work dynamically and periodically during the execution of the program. Since the characterization overhead of decision trees is the lowest, they are an ideal candidate for this application.

In [65], the authors detect potential pipeline and do-all parallel patterns from sequential applications by using template matching technique. However, we detect six parallel pattens including task parallel and geometric decomposition in parallel multi-threaded application.

7.4.2. Machine Learning Techniques

Zanoni et al. [107] developed a tool, MARPLE-DFD, that uses machine learning techniques to detect design patterns. Their experimental results show that the usage of machine learning provides a significant performance improvement. We also show

that detecting parallel patterns provides performance improvements and enables optimizations. Wang et al. [108] use machine learning techniques to partition streaming applications. They reduce the number of the features of an application with PCA. They then use k-means clustering and Bayesian Information Criterion (BIC) to build their model. They run kNN technique on their model to predict the partitioning structure of an application. We use PCA to reduce the number of the sub-characteristics we use for pattern classification similar to their work. The features of an application they use are different from our sub-characteristics and they find ideal partitioning of applications, whereas we find parallel patterns of applications. In [91], the authors use PCA and BIC with k-means clustering to measure the similarity between benchmarks. They cluster 21 benchmarks and find a subset of representative benchmarks using 29 microarchitecture independent characteristics. They find 12 optimum clusters and select one representative benchmark from each cluster. In this thesis, we use only 4 characteristics and find 6 clusters, where each cluster represents a parallel pattern. We do not select representative benchmarks because we use a parallel pattern to describe the characteristics of a cluster. Hence, one can easily interpret the characteristics of a cluster without knowing the characteristics of the representative benchmark of the cluster.

Cavazos et al. [109] collect microarchitecture dependent characteristics by using hardware performance counters and then use these characteristics with machine learning (logistic regression) to predict good compiler optimizations that improve the performance in terms of execution time. Cammarota et al. [110] select inlining vectors for program performance (completion time) optimization by applying machine learning techniques using hardware performance counters. They use 10-fold cross-validation to select the most suitable technique similar to us. In [111], the authors use machine learning to improve the performance and energy efficiency by predicting code transformations. They use a configuration vector (including thread count, software prefetching, and padding) and a performance vector (including cycles per thread, cache misses, and power meter) to represent each application and then use these vectors as inputs to machine learning techniques. In [112, 113], the authors use performance values (completion times) to characterize parallel applications and propose a technique to find

a representative subset of benchmarks used for performance evolution. None of the above techniques use parallel patterns or capture the high level software architectural behaviors captured as general threading, thread communication, and data sharing characteristics by us during their analysis. Ding et al. [114] measure code similarity in terms of code syntax and cost-model provided metrics and then they detect codes that can be optimized (ported) in a similar way without profiling the codes.

8. CONCLUSIONS AND FUTURE WORK

8.1. Summary

Development of new multicore systems including CPUs and GPUs requires a large number of benchmarks where benchmarks capture the essence of many important real-life applications and allow performance, and power analysis. At the same time, there is an increase in simulation times of benchmarks that limits our ability to fully explore the design space. Furthermore, the existing benchmarks can require specific architectures or libraries such as shared memory architectures, or Pthreads, OpenMP, and OpenCL libraries where multicore systems cannot not be able to use these benchmarks as they cannot support such architectures or libraries. In addition, benchmarks or real-life applications can be intellectual property and cannot be shared with developers for design studies.

The solution in this thesis is to develop synthetic benchmarks suitable for any given infrastructure where synthetic benchmarks are a miniaturized form of benchmarks that allow high simulation speeds and act as proxies of proprietary applications. In order to derive these synthetic benchmarks from the original applications, we propose an automatic benchmark synthesis framework with characterization and generation components targeting multicore CPU and GPU systems. To characterize CPU applications, we use parallel software architectural patterns from which the structure of the synthetic benchmark are composed. We use several machine learning techniques, which are often feasible and cost-effective for classification, to detect the parallel pattern used in a multi-threaded application.

We evaluated our framework on four major applications, namely, benchmark synthesis for multicore CPUs, thread-level synthetic benchmarks for multicore CPUs, benchmark synthesis for GPUs, and parallel pattern detection. Each of these contributions are summarized in this section.

Synthetic Benchmark Generation for Multicore CPUs: We present a novel automated multicore benchmark synthesis framework with characterization and generation components. Our framework uses parallel patterns in capturing important characteristics of multi-threaded applications and generates synthetic multicore benchmarks from those applications. The resulting synthetic benchmarks are small, fast, portable, human-readable, and they accurately reflect microarchitecture dependent and independent characteristics of the original multicore applications. Also, they can use either Pthreads or MCA libraries. Thanks to MCA libraries, our benchmarks can be run on any given infrastructure, that is, SMP or message passing, unlike previously developed benchmarks. Hence, this allows us to target heterogeneous embedded multicore systems. We implement our techniques in the MINIME tool and generate synthetic benchmarks from PARSEC, Rodinia, and EEMBC MultiBench benchmarks on x86 and Power Architecture platforms. We show that synthetic benchmarks are representative across a range of multicore machines with different architectures, while being on average 21× faster and 14× smaller than original benchmarks.

Thread-Level Synthetic Benchmarks for Multicore CPUs: We present a novel automated thread-level synthetic benchmark generation framework with characterization and generation components. The resulting thread-level synthetic benchmarks are fast, portable, human-readable, and they accurately mimic the microarchitecture dependent and independent characteristics of each thread in original application. We demonstrate that we can generate multi-threaded synthetic benchmarks for real-life PARSEC and Rodinia benchmarks, while being faster (on average 147×) and smaller (on average 11×) than originals. The obtained results show that synthetic benchmarks not only accurately preserve thread-level microarchitecture dependent and independent characteristics but also parallel programming patterns.

Synthetic Benchmark Generation for GPUs: We introduce a novel automated benchmark synthesis framework for GPUs, called MINIME-GPU, to speed up architectural simulation of modern GPU architectures. Our framework captures important characteristics of original GPU applications and generates synthetic GPU benchmarks using OpenCL library from those applications. To the best of our knowl-

edge, this is the first time synthetic OpenCL benchmarks for GPUs are generated from existing applications. We use several characteristics including instruction throughput, compute unit occupancy, and memory efficiency to compare the similarity of original applications and their corresponding synthetic benchmarks. The experimental results show that our synthetic benchmark generation framework is capable of generating synthetic benchmarks that have similar characteristics with the original applications they are generated from. On average, the similarity (accuracy) is 96% and the speedup is 541×. Also, our synthetic benchmarks use OpenCL library, that allows us to obtain portable human readable benchmarks as opposed to using assembly level code and they are faster and smaller than the original applications that they are generated from. We experimentally validated that our synthetic benchmarks preserve the characteristics of the original applications across different architectures.

Using Machine Learning Techniques to Detect Parallel Patterns of Multi-Threaded Applications: Multicore hardware and software are becoming increasingly more complex. The programmability problem of multicore software has led to the use of parallel patterns. Parallel patterns reduce the effort and time required to develop multicore software by effectively capturing its thread communication and data sharing characteristics. Hence, detecting the parallel pattern used in a multi-threaded application is crucial for performance improvements and enables many architectural optimizations; however, this topic has not been widely studied. We apply machine learning techniques in a novel approach to automatically detect parallel patterns and compare these techniques in terms of accuracy and speed. We experimentally validate the detection ability of our techniques on benchmarks including PARSEC and Rodinia. Our experiments show that the k-nearest neighbor, decision trees, and naive Bayes classifier are the most accurate techniques. Overall, decision trees are the fastest technique with the lowest characterization overhead producing the best combination of detection results. We also show the usefulness of the proposed techniques on synthetic benchmark generation.

In Table 8.1, we summarize the synthetic benchmarks we generate in this thesis. In the table, we show the inputs, outputs, experimental environment, and experimental

Table 8.1. Our synthetic benchmarks with experimental details.

	Synthetic Benchmark		
	Application-level CPU	**Thread-level CPU**	**GPU**
Input	PARSEC, Rodinia (OpenMP), and EEMBC MultiBench suites in C using Pthreads	PARSEC and Rodinia (OpenMP) suites in C using Pthreads	AMD APP SDK benchmarks in C/C++ using OpenCL
Experimental environment	x86 and Power Architecture hardware	x86 hardware	AMD Southern Islands GPUs in Multi2Sim simulator and AMD HD 7950 GPU hardware
Experimental setup	overall similarity threshold is 90%, individual similarity threshold is 80%, and iteration upper bound is 40	individual similarity threshold is 80%, and iteration upper bound is 100	overall similarity threshold is 90%, individual similarity threshold is 60%, and iteration upper bound is 20
Characterization tools	DynamoRIO, Umbra, and Linux perf	DynamoRIO, Umbra, and PapiEx	Multi2Sim simulator
Characteristics	Parallel pattern, thread communication, data sharing, general threading, and performance (IPC, CMR, BMR, and CCR)	Parallel pattern, thread communication, data sharing, and general threading, and per thread performance (IPC, CMR, and BMR)	Instruction throughput, CMAR, dynamic memory instruction mix, memory efficiency, and compute unit occupancy
Output	Synthetic benchmarks in C using Pthreads, MCAPI or MRAPI	Synthetic benchmarks in C using Pthreads, MCAPI or MRAPI	Synthetic benchmarks in C/C++ using OpenCL
Avg. app-level similarity	92%	93%	96%
Avg. thread-level similarity	44%	84%	-
Avg. speedup	21×	147×	541×
Avg. code size reduction	14×	11×	1×
Avg. number of iterations	12	63	7

results including average application-level similarity, thread-level similarity, speedup, and code size reduction for our application-level CPU, thread-level CPU, and GPU synthetic benchmarks. We show the benchmarks we use as input and the properties of the synthetic benchmarks we generate. We list the real hardware and simulators we use in the experimental environment and represent the experimental setup (parameters) including similarity and iteration thresholds. Furthermore, we show the characterization tools we use and we list the characteristics we collect from an application to generate a synthetic benchmark. We represent the average application-level and thread-level similarities where the thread-level similarity is not applicable for GPU benchmarks.

We show the average speedup obtained in terms of execution time and the average reduction in lines of code going from the original to the synthetic. Note that there is no reduction in code size for synthetic GPU benchmarks since original GPU applications are already small. The average number of iterations it takes to generate the synthetic benchmark is also represented in the table.

8.2. Future Work

Synthetic Benchmark Generation for Multicore Mobile Platforms: Mobile devices such as smartphones and tablets are widely used in many domains and the rate of their usage continues to grow. Meanwhile, a diverse set of mobile applications including social networks, document processing, web browsers, and games have been developed. It would be useful to understand the performance and power limitations of these platforms and applications in order to improve the future mobile platforms. Hence, an important direction in extending our work is to develop MINIME-Mobile framework to generate synthetic benchmarks for mobile platforms.

Standalone Synthetic Benchmark Generation for Multicore Systems: It is important to develop MINIME-Standalone framework in which we generate synthetic benchmarks from user defined characteristics including parallel patterns instead of using existing applications. Hence, one can use these user defined synthetic benchmarks to speed up the process of early performance evaluation and architectural exploration studies. Furthermore, once we generate synthetic benchmarks using MINIME-Standalone framework, it is also important to use these synthetic benchmarks to validate new multicore architectures and libraries.

Synthetic Benchmark Generation for Multicore CPUs: We can generate synthetic CPU benchmarks that preserve phase behavior and power consumption characteristics. This is because these characteristics are exploited in various tasks and our synthetic benchmarks can also be used in these tasks once we preserve these characteristics. For example, phase behavior is exploited in and power consumption is used in power usage optimizations and dynamic voltage and frequency scaling tasks. Cur-

rently we use a greedy algorithm to generate synthetic benchmarks however we believe that using genetic algorithms can improve the quality of our synthetic benchmarks. That is, genetic algorithms can generate more accurate synthetic benchmarks in fewer iterations.

Synthetic Benchmark Generation for GPUs: An important direction in extending our work is to perform experiments on other GPU architectures such as Intel and NVIDIA. Also, power consumption and communication characteristics between the host and compute devices are characteristics that we want to preserve in the future. It is important to mimic dwarfs of original GPU applications similar to our technique for CPUs in which we mimic the parallel patterns of original CPU benchmarks. Note that, in this case, we need to detect (recognize) the dwarfs of original applications. In addition, it would be useful to preserve other patterns including prefix and reduction.

Synthetic Benchmark Generation for heterogeneous CPU and GPU systems: In this thesis, we generate synthetic GPU benchmarks that only preserve the characteristics of GPU kernel programs, that is, we do not mimic the characteristics of CPU host programs. This is because, in general, the whole solution of the problem is implemented on the GPU side. However, it would be useful to use MINIME (CPU) to generate synthetic benchmarks for CPU host programs and MINIME-GPU to generate synthetic benchmarks for GPU kernel programs for such CPU/GPU benchmarks in which some parts of a problem are solved on the CPU side and the other parts are solved on the GPU side.

Parallel Pattern Detection: Since most applications have time varying phase behavior, parallel pattern detection techniques can be extended to detect phase behavior, for instance, changing between memory and compute-insensitive phases. In addition, some applications can have a mixture of multiple parallel patterns and it would be useful to detect these patterns in multi-threaded applications. Note that using our techniques for GPUs can guide the development and optimization of parallel architectures as well as novel programming models but also introduces new challenges that include defining a set of important GPU application characteristics, correlating

these characteristics with GPU application patterns (dwarfs), and obtaining a large number of GPU applications to train our techniques. It would be useful to validate the usefulness of our techniques on choosing useful benchmarks for performance evaluation and benchmark subsetting.

REFERENCES

1. Multi2Sim, "*Multi2Sim: A Heterogeneous System Simulator*", 2014, `https://www.multi2sim.org/`, 1 April 2014.

2. Poovey, J. A., B. P. Railing, and T. M. Conte, "Parallel Pattern Detection for Architectural Improvements", *USENIX conference on Hot topic in parallelism (HotPar)*, pp. 1–6, 2011.

3. Bienia, C., S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 72–81, 2008.

4. Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

5. Parboil Benchmark suite, "*Parboil Benchmarks*", 2015, `http://impact.crhc.illinois.edu/Parboil/parboil.aspx`, 1 April 2015.

6. Intel, "*Embedded Microprocessor Benchmark Consortium*", 2013, `http://www.eembc.org`, 1 April 2013.

7. Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", *SIGARCH Computer Architure News*, Vol. 23, No. 2, pp. 24–36, 1995.

8. Bailey, D. H., "NAS Parallel Benchmarks", *Encyclopedia of Parallel Computing*, pp. 1254–1259, NASA Ames Research Center, 2011.

9. Joshi, A., L. Eeckhout, and L. John, "The Return of Synthetic Benchmarks", *SPEC Benchmark Workshop*, pp. 1–11, 2008.

10. Hoste, K. and L. Eeckhout, "Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 83–92, 2006.

11. Ruparelia, N. B., "Software Development Lifecycle Models", *ACM SIGSOFT Software Engineering Notes*, Vol. 35, No. 3, pp. 8–13, 2010.

12. Deshpande, A. and D. Riehle, "The Total Growth of Open Source", *Open Source Development, Communities and Quality*, Vol. 275 of *IFIP - The International Federation for Information Processing*, pp. 197–209, Springer, 2008.

13. Mitchell, T. M., *Machine Learning*, McGraw-Hill, Inc., 1st edition, 1997.

14. Alpaydin, E., *Introduction to Machine Learning*, The MIT Press, 2nd edition, 2010.

15. Che, S. and K. Skadron, "BenchFriend Correlating the Performance of GPU Benchmarks", *International Journal of High Performance Computing Applications (IJHPCA)*, Vol. 28, No. 2, pp. 238–250, 2014.

16. Seo, S., G. Jo, and J. Lee, "Performance Characterization of the NAS Parallel Benchmarks in OpenCL", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 137–148, 2011.

17. Bakhoda, A., G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator", *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163–174, 2009.

18. Kerr, A., G. Diamos, and S. Yalamanchili, "A Characterization and Analysis of PTX Kernels", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 3–12, 2009.

19. Goswami, N., R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, 2010.

20. MCA, "*Multicore Association*", 2013, `http://www.multicore-association.org`, 1 April 2013.

21. 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, 2012.

22. Deniz, E. and A. Sen, "Gomulu Sistemler Icin Karsilastirma Uygulamasi Gelistirme", *Gomulu Sistemler ve Uygulamalari Sempozyumu (GOMSIS)*, pp. 1–6, 2014.

23. Deniz, E., A. Sen, B. Kahne, and J. Holt, "MINIME: Pattern-Aware Multicore Benchmark Synthesizer", *IEEE Transactions on Computers (TC)*, Vol. 64, No. 8, pp. 2239–2252, 2015.

24. Sen, A. and E. Deniz, "Thread-level Synthetic Benchmarks for Multicore Systems", *Microprocessors and Microsystems*, Vol. 39, No. 7, pp. 471 – 479, 2015.

25. Deniz, E. and A. Sen, "MINIME-GPU: Multicore Benchmark Synthesizer for GPUs", *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 12, No. 4, pp. 34:1 – 34:25, 2015.

26. Deniz, E. and A. Sen, "Using Machine Learning Techniques to Detect Parallel Patterns of Multi-threaded Applications", *International Journal of Parallel Programming (IJPP)*, Vol. OnlinePublished, No. 1, pp. 1 – 34, 2015.

27. Ortega-Arjona, J. L. and G. Roberts, "Architectural Patterns for Parallel Programming", *European Conference on Pattern Languages of Programs (Euro-*

PLoP), pp. 225–260, 1998.

28. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1st edition, 1994.

29. Keutzer, K., B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects", *Workshop on Parallel Programming Patterns (ParaPLoP)*, pp. 9:1–9:8, 2010.

30. Mattson, T., B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005.

31. MPI, "*The Message Passing Interface (MPI) standard*", 2013, `http://www.mcs.anl.gov/research/projects/mpi/`, 1 April 2013.

32. Khronos OpenCL Working Group, "*The OpenCL specification version 1.2*", 2012, `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, 14 November 2012.

33. Dunteman, G. H., *Principal Component Analysis*, Sage Publications, 1989.

34. Jain, A. K. and R. C. Dubes, *Algorithms for clustering data*, Prentice-Hall, Inc., 1988.

35. John, L. K. and L. Eeckhout, *Performance evaluation and benchmarking*, CRC Press, 2005.

36. SPEC CPU2006, "*SPEC - Standard Performance Evaluation Corporation*", 2013, `http://www.spec.org/cpu2006/`, 1 April 2013.

37. Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark

suite", *IEEE International Workshop/Symposium on Workload Characterization (IISWC)*, pp. 3–14, IEEE, 2001.

38. Holt, J., A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister, "Software Standards for the Multicore Era", *IEEE Micro*, Vol. 29, No. 3, pp. 40–51, 2009.

39. Joshi, A., L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 105–115, 2006.

40. 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.

41. Barrow-Williams, N., C. Fensch, and S. Moore, "A communication characterisation of Splash-2 and Parsec", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 86–97, 2009.

42. The Open Group, "*POSIX Standard, IEEE Std 1003.1, 2013 Edition*", 2014, `http://www.unix.org/version4/ieee\_std.html`, 1 April 2014.

43. OpenMP, "*The OpenMP API Specification for Parallel Programming*", 2013, `http://openmp.org`, 1 April 2013.

44. DynamoRio, "*DynamoRIO Dynamic Instrumentation Tool Platform*", 2013, `http://dynamorio.org/`, 1 April 2013.

45. 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", *Programming Language Design and Implementation (PLDI)*, pp. 190–200, 2005.

46. Zhao, Q., D. Bruening, and S. Amarasinghe, "Umbra: Efficient and Scalable Memory Shadowing", *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 22–31, 2010.

47. Linux, "*Linux profiling with performance counters*", 2013, `https://perf.wiki.kernel.org`, 1 April 2013.

48. PapiEx, "*PapiEx - Command line/library utility to measure hardware performance counters with PAPI*", 2013, `http://icl.cs.utk.edu/~mucci/papiex/`, 1 April 2013.

49. PAPI, "*Performance Application Programming Interface (PAPI)*", 2013, `http://icl.cs.utk.edu/papi/`, 1 April 2013.

50. Yu, Z., L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C. Xu, and J. Wu, "GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation", *IEEE Transactions on Computers*, Vol. 64, No. 11, pp. 3153–3166, 2015.

51. AMD, "*AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK)*", 2014, `http://developer.amd.com/sdks/amdappsdk/`, 1 April 2014.

52. AMD, "*AMD Southern Island Instruction Set Architecture*", 2014, `http://developer.amd.com/wordpress/media/2012/12/AMD\_Southern\_Islands\_Instruction\_Set\_Architecture.pdf`, 1 April 2014.

53. MATLAB, "*MATLAB: The Language of Technical Computing - MathWorks*", 2014, `http://www.mathworks.com/products/matlab/`, 1 April 2014.

54. Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

55. AMD CodeXL, "*AMD CodeXL: Powerful Debugging, Profiling, and Analysis*", 2015, `http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/`, 1 April 2015.

56. Eeckhout, L., H. Vandierendonck, and K. De Bosschere, "Quantifying the Impact of Input Data Sets on Program Behavior and its Applications", *Journal of Instruction-Level Parallelism (JILP)*, Vol. 5, pp. 1–33, 2003.

57. Bird, S., A. Phansalkar, L. K. John, A. Mercas, and R. Idukuru, "Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture based processor", *SPEC Benchmark Workshop*, pp. 1–7, 2007.

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

59. Jason A. Poovey, T. M. C., Michael C. Rosier, "Pattern-Aware Dynamic Thread Mapping Mechanisms for Asymmetric Manycore Architectures", Technical Report 2011-1, School of Computer Science, Georgia Institute of Technology, Atlanta, 2011.

60. Lin, C.-Y., C.-B. Kuan, W.-L. Shih, and J. Lee, "Compilers for Low Power with Design Patterns on Embedded Multicore Systems", *Journal of Signal Processing Systems (JSPS)*, Vol. 80, No. 3, pp. 277–293, 2015.

61. McCool, M. D., "Structured Parallel Programming with Deterministic Patterns", *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism (HotPar)*, pp. 1–6, 2010.

62. FastFlow: programming multicore, "*FastFlow: Pattern-based multi/many-core parallel programming framework*", 2014, `http://sourceforge.net/projects/mc-fastflow/`, 1 April 2014.

63. Aldinucci, M., S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Design Patterns Percolating to Parallel Programming Framework Implementation", *International Journal of Parallel Programming (IJPP)*, Vol. 42, No. 6, pp. 1012–1031, 2014.

64. Hammond, K., M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer, "The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems", *Formal Methods for Components and Objects*, Vol. 7542 of *Lecture Notes in Computer Science*, pp. 218–236, Springer Berlin Heidelberg, 2013.

65. Huda, Z. U., A. Jannesari, and F. Wolf, "Using Template Matching to Infer Parallel Design Patterns", *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 11, No. 4, pp. 64:1–64:21, 2015.

66. DiscoPoP (Discovery of Potential Parallelism), "*DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities*", 2014, `http://www.grs-sim.de/research/parallel-programming/multicore-programming/discopop-project.html`, 1 April 2014.

67. Campa, S., M. Danelutto, M. Goli, H. González-Vélez, A. M. Popescu, and M. Torquati, "Parallel Patterns for Heterogeneous CPU/GPU Architectures: Structured Parallelism from Cluster to Cloud", *Future Generation Computer Systems (FGCS)*, Vol. 37, pp. 354–366, 2014.

68. Mitchell, T. M., "The Discipline of Machine Learning", Technical Report CMU-ML-06-108, Machine Learning Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2006.

69. John, G. and P. Langley, "Estimating Continuous Distributions in Bayesian Classifiers", *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 338–345, 1995.

70. Moller, M. F., "A scaled conjugate gradient algorithm for fast supervised learning", *Neural Networks*, Vol. 6, No. 4, pp. 525–533, 1993.

71. Bishop, C. M., *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer-Verlag New York, Inc., 2006.

72. Battiti, R., "Using mutual information for selecting features in supervised neural net learning", *IEEE Transactions on Neural Networks (NN)*, Vol. 5, pp. 537–550, 1994.

73. Goswami, D., A. Singh, and B. R. Preiss, "Advances in Software Engineering", *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*, chapter Building parallel applications using design patterns, pp. 243–265, Springer-Verlag New York, Inc., 2002.

74. Zandifar, M., M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing Algorithmic Skeletons to Express High-Performance Scientific Applications", *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, pp. 415–424, 2015.

75. Skillicorn, D. B., "Models for Practical Parallel Computation", *International Journal of Parallel Programming (IJPP)*, Vol. 20, No. 2, pp. 133–158, 1991.

76. Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

77. Che, S., J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11, 2010.

78. Poovey, J. A., T. M. Conte, M. Levy, and S. Gal-On, "A Benchmark Characterization of the EEMBC Benchmark Suite", *IEEE Micro*, Vol. 29, No. 5, pp. 18–29, 2009.

79. Hillenbrand, D., J. Tao, and M. Balzer, "ALPS: A Methodology for Application-Level Communication Characterization of Parsec 2.1", *Procedia Computer Science*, Vol. 4, pp. 2086–2095, 2011.

80. Bharathi, S., A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of Scientific Workflows", *Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pp. 1 –10, 2008.

81. Oskin, M., F. T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs", *International Symposium on Computer Architecture*, pp. 71–82, 2000.

82. Nussbaum, S. and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation", *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 15–24, 2001.

83. Eeckhout, L., R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies", *International Symposium on Computer Architecture (ISCA)*, pp. 350–361, 2004.

84. Ertvelde, L. V. and L. Eeckhout, "Benchmark Synthesis for Architecture and Compiler Exploration", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11, 2010.

85. Hughes, C. and T. Li, "Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 163–172, 2008.

86. Kim, K., C. Lee, J. H. Jung, and W. W. Ro, "Workload Synthesis: Generating Benchmark Workloads from Statistical Execution Profile", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 120–129, 2014.

87. Ferrari, D., "On the Foundations of Artificial Workload Design", *SIGMETRICS Performance Evaluation*, Vol. 12, No. 3, pp. 8–14, 1984.

88. Wang, Y. and Y. Solihin, "Emulating Cache Organizations on Real Hardware Using Performance Cloning", *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 298–307, 2015.

89. Ittershagen, P., P. A. Hartmann, K. Grüttner, and W. Nebel, "A Workload Extraction Framework for Software Performance Model Generation", *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, pp. 3:1–3:6, 2015.

90. Sen, A., G. Kara, E. Deniz, and S. Niar, "Fast System Level Benchmarks for Multicore Architectures", *Euromicro Conference on Digital System Design (DSD)*, pp. 635–638, 2014.

91. Joshi, A., A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring Benchmark Similarity Using Inherent Program Characteristics", *IEEE Transactions on Computers (TC)*, Vol. 55, pp. 769–782, 2006.

92. Jo, J., L. K. John, M. Reese, and J. Holt, "Validation of Synthetic Benchmarks by Measurement", *Workshop on Unique Chips and Systems (UCAS)*, pp. 1–6, 2010.

93. John, L. K., J. Jo, , and K. Ganesan, "Workload Synthesis for a Communications SoC", *In Workshop on SoC Architecture, Accelerators and Workloads, held in conjunction with HPCA-17*, pp. 1–8, 2011.

94. Deshpande, V., X. Wu, and F. Mueller, "Auto-generation of Communication Benchmark Traces", *SIGMETRICS Performance Evaluation Review*, Vol. 40,

No. 2, pp. 99–105, 2012.

95. Jung, C., S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: Effective Selection of Data Structures", *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 86–97, 2011.

96. Sembrant, A., D. Black-Schaffer, and E. Hagersten, "Phase Behavior in Serial and Parallel Applications", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 47–58, 2012.

97. Perelman, E., M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting Phases in Parallel Applications on Shared Memory Architectures", *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–10, 2006.

98. Sherwood, T., E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior", *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 45–57, 2002.

99. Dey, T., W. Wang, J. Davidson, and M. Soffa, "Characterizing Multi-threaded Applications based on Shared-Resource Contention", *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 76–86, 2011.

100. Kerr, A., G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU Workloads and Systems", *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pp. 31–42, 2010.

101. Ukidave, Y., F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "NUPAR: A Benchmark Suite for Modern GPU Architectures", *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pp. 253–264, 2015.

102. Joshi, A. M., L. Eeckhout, R. H. Bell, 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, pp. 10:1–10:33, 2008.

103. Huang, J.-C., L. Nai, H. Kim, and H.-H. Lee, "TBPoint: Reducing Simulation Time for Large-Scale GPGPU Kernels", *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 437–446, 2014.

104. Lee, S. and W. W. Ro, "Parallel GPU architecture simulation framework exploiting work allocation unit parallelism", *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 107–117, 2013.

105. Matsumoto, K., N. Nakasato, and S. G. Sedukhin, "Implementing a Code Generator for Fast Matrix Multiplication in OpenCL on the GPU", *IEEE International Symposium on Embedded Multicore Socs (MCSoC)*, pp. 198–204, 2012.

106. Che, S., J. Meng, and K. Skadron, "Dymaxion++: A Directive-Based API to Optimize Data Layout and Memory Mapping for Heterogeneous Systems", *IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, pp. 916–924, 2014.

107. Zanoni, M., F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection", *Journal of Systems and Software (JSS)*, Vol. 103, pp. 102 – 117, 2015.

108. Wang, Z. and M. F. P. O'boyle, "Using Machine Learning to Partition Streaming Programs", *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 10, No. 3, pp. 20:1–20:25, 2008.

109. Cavazos, J., G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly Selecting Good Compiler Optimizations using Performance Counters", *International Symposium on Code Generation and Optimization (CGO)*, pp. 185–197, 2007.

110. Cammarota, R., A. Nicolau, A. V. Veidenbaum, A. Kejariwal, D. Donato, and M. Madhugiri, "On the Determination of Inlining Vectors for Program Optimization", *Compiler Construction*, pp. 164–183, 2013.

111. Ganapathi, A., K. Datta, A. Fox, and D. Patterson, "A Case for Machine Learning to Optimize Multicore Performance", *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, pp. 1–6, 2009.

112. Cammarota, R., L. A. Beni, A. Nicolau, and A. V. Veidenbaum, "Effective Evaluation of Multi-core Based Systems", *International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 19–25, 2013.

113. Cammarota, R., A. Kejariwal, P. D'Alberto, S. Panigrahi, A. V. Veidenbaum, and A. Nicolau, "Pruning Hardware Evaluation Space via Correlation-Driven Application Similarity Analysis", *Proceedings of the 8th ACM International Conference on Computing Frontiers (FC)*, pp. 4:1–4:10, 2011.

114. Ding, W., O. Hernandez, T. Curtis, and B. Chapman, "Porting Applications with OpenMP Using Similarity Analysis", *Languages and Compilers for Parallel Computing (LCPC)*, pp. 20–35, Springer, 2014.