Microprocessors and Microsystems 39 (2015) 471–479

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Thread-level synthetic benchmarks for multicore systems

Alper Sen *, Etem Deniz

Department of Computer Engineering, Bogazici University, Istanbul, Turkey

ARTICLE INFO

Article history:
Available online 29 July 2015

Keywords:
Multicore systems
Parallel patterns
Synthetic benchmarks

ABSTRACT

One of the commonly used techniques to speedup early architectural exploration and performance evaluation of new hardware architectures is to use synthetic benchmarks. This paper presents 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 micro-architecture 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 micro-architecture dependent and independent characteristics but also parallel programming patterns, which are high-quality solutions to frequently occurring problems in parallel programming.

1. Introduction

Benchmarks are tests of computer systems that help estimate performance of a system on a workload, hence optimize designs. 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 allowed design time.

One of the commonly used techniques to boost the performance of simulations is to use *synthetic benchmarks*. These benchmarks can either be derived from existing benchmarks 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 benchmark that it is derived from so that it simulates faster.

The emergence of multi-core systems has made parallel benchmark suites ubiquitous. These new generation of benchmarks allow to adequately evaluate the future parallel computer architectures such as multi-cores, many-cores, accelerators, and supercomputers. Some of the commonly used parallel benchmark suites are PARSEC [1], Rodinia [2], SPLASH-2 [3], and NAS Parallel Benchmarks [4], all of which suffer from slow speeds when run on simulators. In this work, we generate synthetic multi-threaded benchmarks based on existing multi-threaded benchmark suites such as PARSEC and Rodinia. Our synthetic benchmarks should be used for early architectural exploration and performance evaluation. The need for more accurate performance evaluation in later design cycles still necessitates the usage of original existing multi-threaded benchmark suites.

We previously developed the MINIME tool [5] that can generate synthetic benchmarks for multicore systems. We obtained small and fast benchmarks upon using MINIME. However, the accuracy of our earlier synthetic benchmarks were low since we did not preserve the characteristics of individual threads in synthetic benchmarks. In that work, we used 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 were added to the main function of the synthetic benchmark. We call those earlier synthetics as *application-level synthetics*. In this work, 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

* Corresponding author. Tel.: +90 212 359 7696.
E-mail addresses: alper.sen@boun.edu.tr (A. Sen), etem.deniz@boun.edu.tr (E. Deniz).

http://dx.doi.org/10.1016/j.micpro.2015.07.010
0141-9331/

application-level synthetics, where an aggregate code block is added to the main function only.

We exploit the concept of *parallel programming patterns* in order to accurately capture the characteristics of parallel applications in synthetic benchmarks. Parallel patterns [6] are generalized, high-quality solutions to frequently occurring problems in parallel programming. These patterns capture thread communication and data sharing characteristics of multi-threaded applications while simplifying multi-threaded application development by raising the abstraction level. Parallel patterns allow us to obtain high quality synthetics as will be shown in the experiments. We developed a new decision tree based pattern recognition algorithm with lower characterization overhead and faster speed than earlier k-nearest neighbor based approaches.

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.

The paper is organized as follows. We present the related work in the next section. We describe our synthetic benchmark development framework in Section 3. We compare application-level and thread-level synthetics in Section 4. Section 5 shows our detailed experimental results. This is followed by the conclusions.

2. Related work

We use micro-architecture independent characteristics in this work which are similar to the characteristics in Poovey et al. [7]. They detect parallel patterns of PARSEC and SPLASH-2 benchmarks using kNN technique with 50% accuracy. Also, in [8], the authors use parallel patterns for dynamic thread mapping based on thread behaviors. We use decision tree technique and our pattern recognition accuracy is 90% for PARSEC and Rodinia in this work.

There has been several works to measure and optimize the performance of benchmarks by using performance counters [9–11]. Micro-architecture dependent characteristics such as cycles per instruction, cache miss rate, and branch misprediction rate are used in these works. These studies are helpful in understanding performance and finding bottlenecks. Hoste et al. [12] introduced micro-architecture independent characteristics to generate benchmarks.

In [13–16], statistical simulation is used to generate synthetic benchmarks. Ertvelde et al. [16] use a micro-architecture dependent way of modeling memory access behavior to generate single-threaded synthetic benchmarks in C. In [17], 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. In [18], 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. Wang and Solihin [19] 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 [20], 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.

There exist many studies on synthetic benchmarks for single-threaded applications [21,22]. However, there are few recent studies for generating synthetics for multi-threaded applications [11,23,24,17]. Their synthetics use low level assembly language, whereas we generate multi-threaded benchmarks as readable and portable C code. Also, their benchmarks are for general purpose multicore systems while we can use MCA libraries and support embedded multicore systems as well as general purpose multicore systems. Our technique runs on real multicore hardware to gather thread-level characteristics and faster than their approach since they use simulators. We also use fewer characteristics (4) compared to their characteristics (around 40), which improve the performance of our technique. Additionally, we use the concept of parallel patterns that capture inherent characteristics and thread behaviors of a multi-threaded application. In [25], we generated synthetic benchmarks in C using MCA libraries for embedded multicore systems. We extended this work in [5] and added support for Pthread library and generated synthetic benchmarks with better application-level similarity. Different from all these earlier works, we use thread-level synthetic benchmarks as opposed to application-level synthetics in this paper.

3. Thread-level synthetic benchmark development framework

Fig. 1 shows the architecture of our MINIME tool [5], which we adapted for this work. The tool has three modules, namely, *benchmark characterizer, parallel pattern classifier*, and *benchmark generator*, where the benchmark characterizer captures the important characteristics 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. We will now explain these components.

3.1. Benchmark characterizer

The benchmark characterizer module collects characteristics of the original application using a combination of dynamic binary instrumentation and performance monitoring counters for micro-architecture independent and micro-architecture dependent characteristics, respectively.

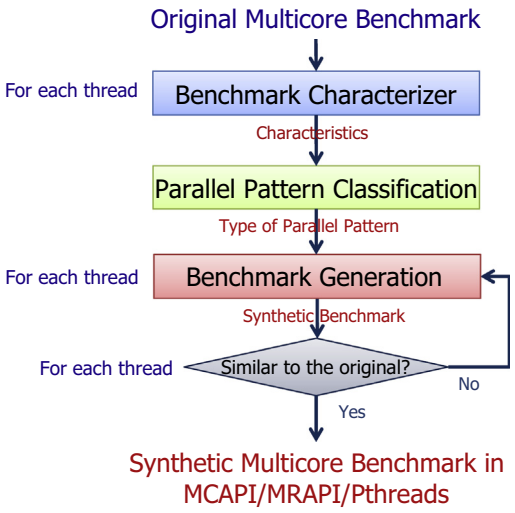

Fig. 1. MINIME: multi-threaded benchmark development framework [5].

We use DynamoRIO [26] dynamic binary instrumentation tool for gathering characteristics during the execution of a benchmark. Binary instrumentation allows us to work with legacy/proprietary IPs and not compile/link applications. We collect general threading characteristics (program counter, dynamic instruction count, creator, lifetime), thread communication characteristics (communicating threads, communication volume), and data sharing characteristics (private, read-only, producer/consumer, migratory). Next we briefly explain these characteristics.

We collect creation and exit times of threads and calculate the lifetime sub-characteristics of threads. We use the program counter sub-characteristic to indicate the starting program counter of a thread. Note that program counter allows us to determine whether the threads are executing the same task or not. We define the ratio of communicating threads as (the number of communicating threads/the total number of threads) and the ratio of communication volume as (the number of cachelines used in communication/the total number of cachelines used).

We use hardware performance counters for collecting micro-architecture dependent characteristics. In the previous version of MINIME, we used Linux perf tools [27] 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 added support for PapiEx tool [28] in this work. PapiEx is a command line utility to measure per-thread and application level hardware performance counters with Performance Application Programming Interface (PAPI) [29]. 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.

3.2. Parallel pattern classifier

Parallel patterns are generalized, high-quality solutions to frequently occurring problems. These patterns describe the problem, components, forces, and solution. Developers can understand parallel patterns quickly and use them while implementing new multicore applications. Parallel patterns are not only applicable for applications that run on CPUs but also for applications that run on GPUs or other platforms. From the view of algorithm structure pattern, a multi-threaded application can be classified as *Task Parallel (TP)*, *Divide and Conquer (DaC)*, *Geometric Decomposition (GD)*, *Recursive Data (RD)*, *Pipeline (Pl)*, and *Event-based Coordination (EbC)* as formally described in [6].

Each pattern type defines a particular behavior for each thread. For example, for task parallel applications, all threads can execute different functions concurrently and can have different lifetimes. In divide and conquer pattern, problems are divided into smaller sub-problems where each thread solves a sub-problem and merges these results. Mergesort is an example of divide and conquer parallel pattern. In task parallel and divide and conquer parallel patterns, shared data is usually read-only and threads also work on private data. We have defined similar behaviors for other parallel pattern types.

In this work, we use decision trees technique to determine the parallel pattern of a given application. We describe this technique in the experiments section.

3.3. Benchmark generator

Finally, the benchmark generator module generates a multi-threaded synthetic benchmark using the parallel pattern type of the application and the performance counter values. The synthetic benchmark consists of a main function and a function for each thread. 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.

We now define formally our similarity metrics for each of the IPC, CMR, and BMR characteristics. We first define $error\_rate = (value\ of\ synthetic - value\ of\ original)/(value\ of\ original)$ and $similarity = (1 - error\_rate) * 100$. We calculate sim_IPC, sim_CMR, and sim_BMR by using *similarity* formula. We define Thread Similarity Score (TSS_i) for each thread i in a benchmark as follows. $TSS_i = (sim\_IPC_i + sim\_CMR_i + sim\_BMR_i)/3$, where sim_IPC_i is the IPC similarity of thread i. Similarly, given n threads, we define average TSS as $aTSS = (TSS_1 + \cdots + TSS_n)/n$.

In our earlier work [5], we defined Overall Similarity Score (*OSS*), which is an application-level similarity score, as $OSS = (sim\_IPC + sim\_CMR + sim\_BMR)/3$, where sim_IPC is the IPC similarity of the application. As we will show in the next section, having a high *OSS* value does not mean having high *TSS* values. Whereas, having a high *TSS* value not only implies having a high *OSS* value but also a more accurate synthetic.

Our synthetic benchmarks use C language. Since C is a high level programming language, synthetics are portable and human-readable as opposed to previous work that use non-portable low level assembly code [22]. Synthetics can use a multitude of parallel libraries. For example, benchmarks using POSIX Pthreads API are created for general purpose multicore systems, whereas benchmarks using MCA APIs (MCAPI or MRAPI) [30] are generated for heterogeneous embedded systems.

3.3.1. Code blocks

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 in Fig. 2 either to increment or decrement IPC of synthetic benchmark. Similarly, we add code blocks for CMR and BMR metrics.

4. 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.

```
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 }
```

Fig. 2. Code block to increment/decrement IPC.

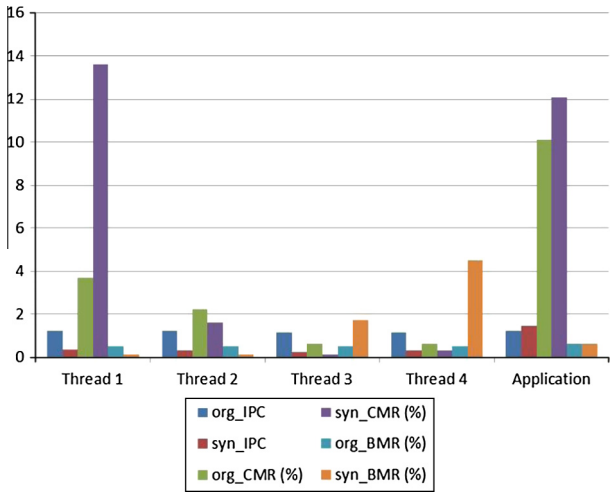

Fig. 3. Characteristics of Blackscholes benchmark and its application-level synthetic.

Fig. 3 shows 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.

Fig. 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.

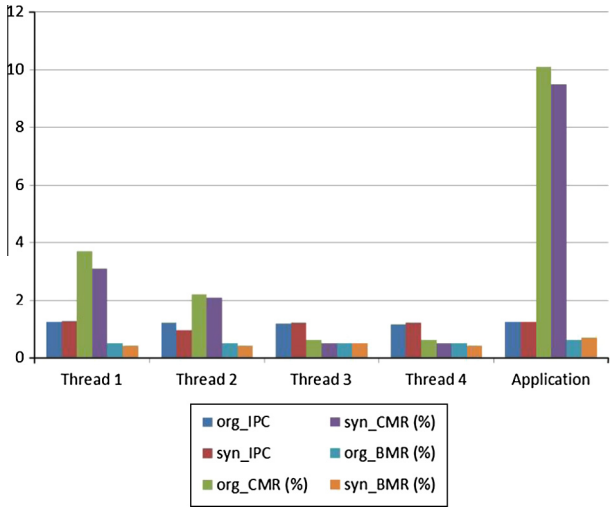

Fig. 4. Characteristics of Blackscholes benchmark and its thread-level synthetic.

The advantages of using our new thread-level synthetic benchmark generation technique over our earlier 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 earlier work as will be shown in the experiments. On the other hand, using our new technique over our earlier technique has some potential downsides. Since we add code blocks to each thread instead of adding them only to the main thread as we did earlier, our thread-level synthetic benchmarks can be larger than our application-level synthetic benchmarks in terms of lines of code. Similarly, generating thread-level synthetic benchmarks requires more iterations because collecting and preserving per thread characteristic is more costly.

5. Experiments

We performed experiments to validate our thread-level synthetic generation technique implemented in MINIME tool. The tool consists of nearly 10 K lines of C code and 500 lines of Python script. The tool and the benchmark results can be downloaded.[1]

Table 1 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 6 MB 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 the GNU Compiler Collection 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 [1] and Rodinia (OpenMP) [2] benchmarks as original benchmarks and generated synthetic benchmarks that use POSIX Pthreads, MRAPI, or MCAPI parallel libraries. PARSEC is a well-known, open-source multi-threaded benchmark with fundamental parallelism constructs. Rodinia is a benchmark suite for heterogeneous computing and cover a wide range of parallel communication patterns, synchronization techniques and power consumption. 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 2 shows benchmark characteristics and our pattern classification results. The column *Known* shows the pattern of the benchmark known from the literature and column *Classified* shows the pattern of the benchmark we classified. Note that parallel patterns of Rodinia benchmarks are not known from the literature, hence we manually decide them from the source code and display these results. 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 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 but due to lack of space we only show Pthreads results here. We also show results for application-level benchmarks from [5] for comparison

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

Table 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

Table 2
Benchmark characteristics and pattern classification results (Task Parallel (TP), Divide and Conquer (DaC), Geometric Decomposition (GD), Recursive Data (RD), Pipeline (Pl), and Event-based Co-ordination (EbC)).

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	20,275	5	TP	TP
	Ferret	10,765	18	Pl	Pl
	Fluidanimate	2784	4	GD	GD
	Swaptions	1095	4	TP	TP
	X264	38,546	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

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(x)* shows the speedup in terms of running time and the column *aTSS* shows the average thread similarity score for the given benchmark.

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 s) 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.

Fig. 5 shows the micro-architecture dependent characteristics of `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. *TSSs* of threads are 85, 95, 93, 93, 89, 88, 85, and 86, respectively. The average TSS *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 Fig. 4. Next, we show the detailed similarity scores for all benchmarks.

In Fig. 6, we give average *TSSs* for all benchmarks and we show the minimum and maximum *TSSs* as error bars. The maximum *aTSS* is 91 in `CFD Solver` and the average of all *aTSSs* 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 micro-architecture 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 previous application-level synthetic benchmarks. We use

Table 3
Thread-level synthetic benchmark generation results.

Original benchmark		Application-level				Thread-level			
Suite	Benchmark	#iter	LOC	Speedup(x)	aTSS	#iter	LOC	Speedup(x)	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	10,174	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

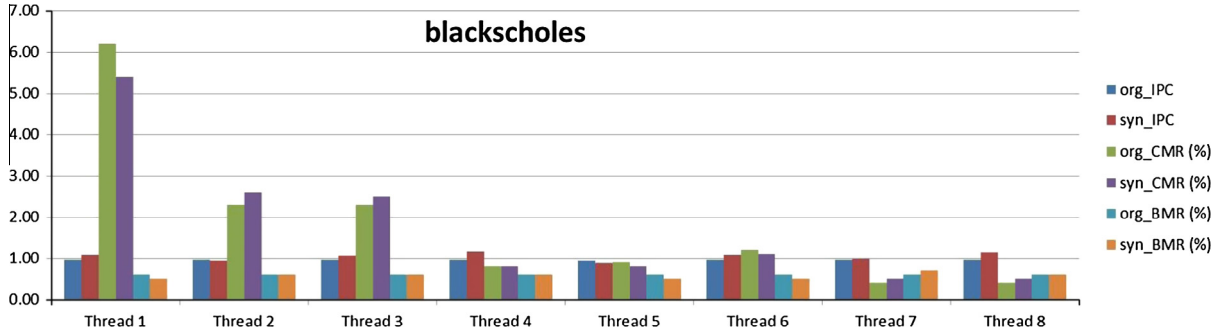

Fig. 5. Characteristics of Blackscholes benchmark and thread-level synthetic of Blackscholes with 8 threads.

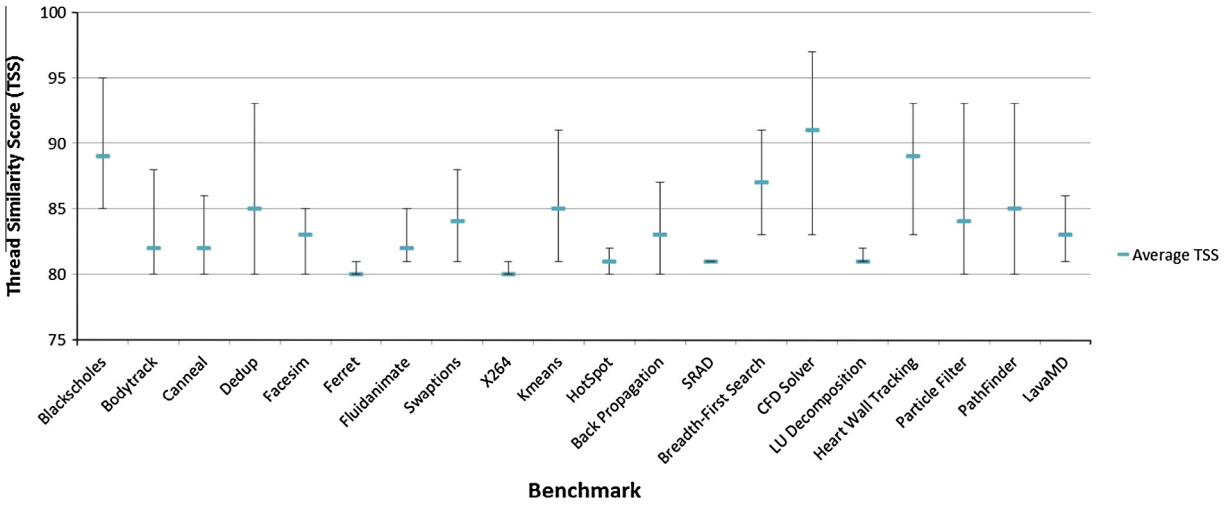

Fig. 6. Average, maximum, and minimum thread similarity scores of all thread-level synthetic benchmarks.

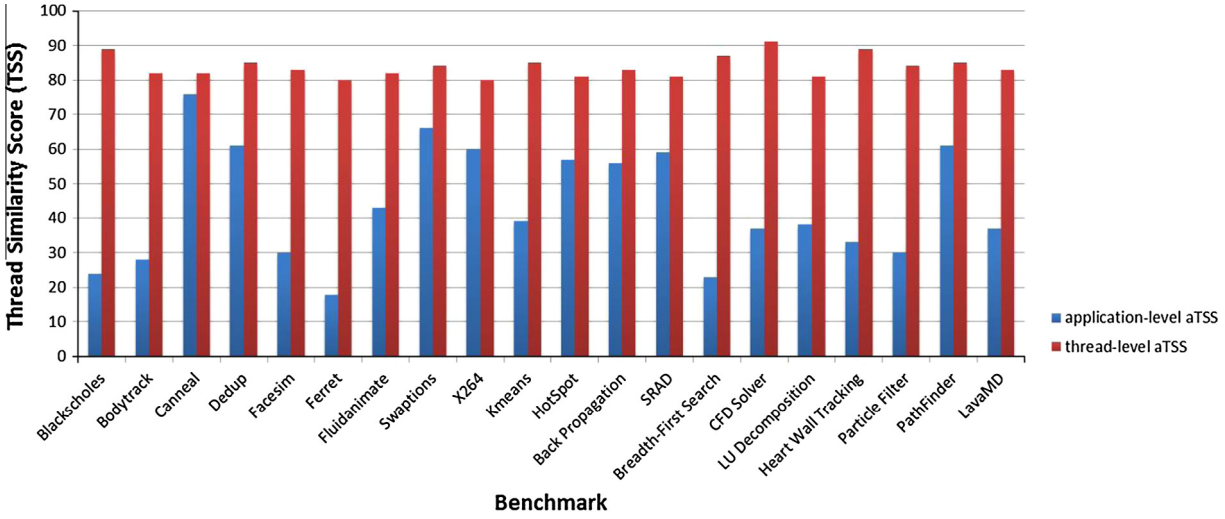

Fig. 7. Comparison of average TSS for application-level and thread-level synthetic of all benchmarks.

the average thread similarity scores (aTSS) of both types of synthetic benchmarks for comparison. We ran our new characterizer on the application-level synthetics that were previously generated in order to collect their thread-level characteristics. Fig. 7 shows our results. It is clear from the figure that thread-level synthetics have much better aTSS (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.

5.1. Decision tree based parallel pattern recognition

In this section we describe an automated way of recognizing parallel patterns described in Section 3.2. As seen from Fig. 1,

automatic and accurate recognition of parallel patterns is crucial for the success of benchmark synthesis.

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. Furthermore, neural networks do not present an easily-understandable classification model. Whereas, decision trees are easy to understand and can be applied easily. We previously developed a parallel pattern recognition technique using the k-nearest neighbor (kNN) technique where we used 12 characteristics with an accuracy of 100% [5]. Given the advantages of decision trees, we wanted to improve the performance of our parallel pattern recognition technique. As we will experimentally show, the decision tree technique requires a fewer number of characteristics compared to the kNN technique. Hence, characterization overhead of the decision tree technique is lower and it runs faster. We now describe the details of using decision trees for pattern recognition.

A decision tree 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.

5.1.1. Construction of a decision tree

We trained our decision tree using the ID3 algorithm [31,32] and obtained the decision tree shown in Fig. 8. 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 [31,32]. Information gain is the reduction in entropy, which characterizes the purity of a subset of the training 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. 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 Program Counter and the test is *Program Counter* < 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 *Program Counter*, *lifetime*, *ratio of communicating threads* and *ratio of communication volume* sub-characteristics. We need only these 4 characteristics 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 ratio of communicating threads or ratio of communication volume 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 4 instead of 12 sub-characteristics can be collected.

In Fig. 9, we show the performance of our decision tree for each parallel pattern as a Receiver Operating Characteristic (ROC) curve.

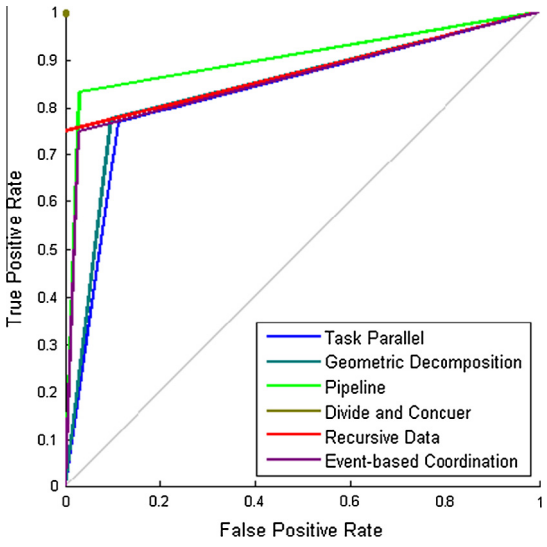

Fig. 9. ROC curve of our decision tree.

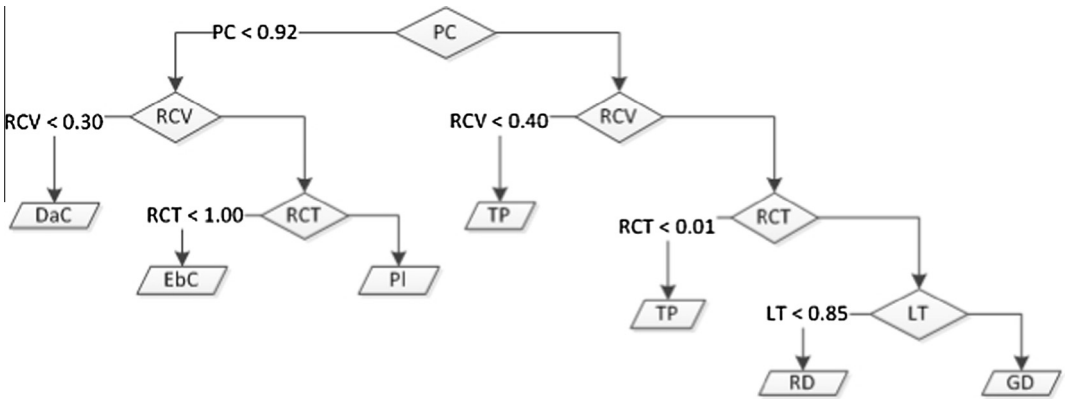

Fig. 8. Decision tree for parallel pattern classification (Characteristics: Lifetime (LT), Program Counter (PC), Ratio of Communicating Threads (RCT), Ratio of Communication Volume (RCV)).

Table 4
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

Table 4 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).

5.1.2. Using the decision tree

We show the parallel pattern classification results of our decision tree in column *Known* of Table 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 s. Whereas, for our earlier pattern recognition results with kNN technique the characterization overhead is 20.2×, since we use all 12 characteristics and the speed is 0.04 s.

5.2. Discussion

Our synthetic benchmarks should be used for early architectural exploration and performance evaluation. The need for more accurate performance evaluation in later design cycles still necessitates the usage of original benchmarks.

Our benchmarks contain code blocks to mimic the behavior of applications. However, one should note that, for example, cache miss rate is very dependent on the size of the cache.

We can use parallel pattern information while sub-setting benchmarks since we know that benchmarks with same parallel pattern have similar high level characteristics. For example, if we know that the parallel pattern of an actual application that will run on the target system is task parallel, then we can use benchmarks which are task parallel to test the target system.

6. Conclusions and future work

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 earlier parallel pattern recognition techniques.

In the future, we plan to expand our work to synthetic benchmark generation for accelerators and investigate other characteristics such as power.

Acknowledgments

This research was supported in part by Semiconductor Research Corporation under task 2082.001, Bogazici University Research Fund 7223, and the Turkish Academy of Sciences.

References

[1] C. Bienia, S. Kumar, J.P. Singh, K. Li, The PARSEC benchmark suite: characterization and architectural implications, in: International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 72–81.
[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: a benchmark suite for heterogeneous computing, in: IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 44–54.
[3] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, SIGARCH Comput. Arch. News 23 (2) (1995) 24–36, http://dx.doi.org/10.1145/225830.223990.
[4] D.H. Bailey, NAS Parallel Benchmarks, in: Encyclopedia of Parallel Computing, NASA Ames Research Center, 2011, pp. 1254–1259.
[5] E. Deniz, A. Sen, B. Kahne, J. Holt, MINIME: pattern-aware multicore benchmark synthesizer, IEEE Trans. Comput. PrePrints (99) (2014). 1–1.
[6] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Addison-Wesley, 2005.
[7] J.A. Poovey, B.P. Railing, T.M. Conte, Parallel pattern detection for architectural improvements, in: USENIX Conference on Hot Topic in Parallelism (HotPar), 2011, pp. 12–12.
[8] J.A. Poovey, M.C. Rosier, T.M. Conte, Pattern-Aware Dynamic Thread Mapping Mechanisms for Asymmetric Manycore Architectures, Tech. Rep. 2011-1, School of Computer Science, Georgia Institute of Technology, 2011.
[9] L. Eeckhout, H. Vandierendonck, K.D. Bosschere, Quantifying the impact of input data sets on program behavior and its applications, J. Instruct.-Level Parallelism 5 (2003) 1–33.
[10] S. Bird, A. Phansalkar, L.K. John, A. Mercas, R. Idukuru, Performance characterization of SPEC CPU benchmarks on intel's core microarchitecture based processor, in: SPEC Benchmark Workshop, 2007, pp. 1–7.
[11] K. Ganesan, L.K. John, V. Salapura, J.C. Sexton, A performance counter based workload characterization on Blue Gene/P, in: International Conference on Parallel Processing (ICPP), 2008, pp. 330–337.
[12] K. Hoste, L. Eeckhout, Comparing benchmarks using key microarchitecture-independent characteristics, in: IEEE International Symposium on Workload Characterization (IISWC), 2006, pp. 83–92.
[13] M. Oskin, F.T. Chong, M. Farrens, HLS: combining statistical and symbolic simulation to guide microprocessor designs, in: International Symposium on Computer Architecture, 2000, pp. 71–82.
[14] S. Nussbaum, J.E. Smith, Modeling superscalar processors via statistical simulation, in: International Conference on Parallel Architectures and Compilation Techniques, 2001, pp. 15–24.
[15] L. Eeckhout, R.H. Bell Jr., B. Stougie, K. De Bosschere, L.K. John, Control flow modeling in statistical simulation for accurate and efficient processor design studies, in: International Symposium on Computer Architecture, 2004, pp. 350–361.
[16] L.V. Ertvelde, L. Eeckhout, Benchmark synthesis for architecture and compiler exploration, in: IEEE International Symposium on Workload Characterization (IISWC), 2010, pp. 1–11.
[17] K. Ganesan, L.K. John, Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors, IEEE Trans. Comput. 63 (4) (2014) 833–846.
[18] K. Kim, C. Lee, J.H. Jung, W.W. Ro, Workload synthesis: generating benchmark workloads from statistical execution profile, in: IEEE International Symposium on Workload Characterization (IISWC), 2014, pp. 120–129.
[19] Y. Wang, Y. Solihin, Emulating cache organizations on real hardware using performance cloning, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 298–307.
[20] P. Ittershagen, P.A. Hartmann, K. Grüttner, W. Nebel, A workload extraction framework for software performance model generation, in: Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), 2015, pp. 3:1–3:6.
[21] A. Joshi, L. Eeckhout, R.H. Bell Jr., L.K. John, Performance cloning: a technique for disseminating proprietary applications as benchmarks, in: IEEE International Symposium on Workload Characterization (IISWC), 2006, pp. 105–115.
[22] A. Joshi, L. Eeckhout, L. John, The return of synthetic benchmarks, in: 2008 SPEC Benchmark Workshop, 2008, pp. 1–11.
[23] C. Hughes, T. Li, Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis, in: IEEE International Symposium on Workload Characterization (IISWC), 2008, pp. 163–172.
[24] K. Ganesan, L.K. John, MAximum Multicore POwer (MAMPO) – an automatic multithreaded synthetic power virus generation framework for multicore systems, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11), 2011, pp. 53:1–53:12.
[25] E. Deniz, A. Sen, J. Holt, B. Kahne, Using software architectural patterns for synthetic embedded multicore benchmark development, in: IEEE International Symposium on Workload Characterization (IISWC), 2012, pp. 89–99.
[26] DynamoRIO Dynamic Instrumentation Tool Platform, 2014. <http://dynamorio.org/>.

[27] Linux Profiling with Performance Counters, 2014. <https://perf.wiki.kernel.org>.
[28] PapiEx – Command Line/library Utility to Measure Hardware Performance Counters with PAPI, 2014. <http://icl.cs.utk.edu/mucci/papiex/>.
[29] Performance Application Programming Interface (PAPI), 2014. <http://icl.cs.utk.edu/papi/>.
[30] Multicore Association, 2014. <http://www.multicore-association.org>.
[31] T.M. Mitchell, Machine Learning, first ed., McGraw-Hill, Inc., New York, NY, USA, 1997.
[32] E. Alpaydin, Introduction to Machine Learning, second ed., The MIT Press, 2010.

Alper Sen received the B.S. and M.S. degrees in electrical and electronics engineering from Middle East Technical University, Ankara, Turkey, in 1995 and 1997, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Texas at Austin, Austin, in 2004. He is currently an Associate Professor with the Department of Computer Engineering, Bogazici University. He was a Technical Staff Member with Freescale Semiconductor, Austin, and an Adjunct Faculty Member with the University of Texas at Austin, until 2009. His current research interests include verification of hardware and software systems, parallel programming, embedded systems, and system-level designs.

Etem Deniz received the B.S. degree in computer engineering from Dokuz Eylul University, Izmir, Turkey, in 2009, and the M.S. degree in computer engineering from Bogazici University, Istanbul, in 2011. He is currently a Ph.D. student with the Department of Computer Engineering, Bogazici University and a software engineer at TUBITAK BILGEM. His current research interests include multicore systems, performance evaluation, software architectural patterns, real-time systems, embedded operating systems, and verification of software systems.