MINIME: Pattern-Aware Multicore Benchmark Synthesizer

Etem Deniz, *Member, IEEE*, Alper Sen, *Senior Member, IEEE*, Brian Kahne, Jim Holt, *Senior Member, IEEE*

Abstract—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. 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 21x faster and 14x smaller than original benchmarks.

Index Terms—Multicore Systems, Parallel Patterns, Synthetic Benchmarks.

1 Introduction

Thermal and power problems limit the performance that single core processors can deliver. This has led to the development of multicore systems. There is a need for efficient parallel programming to fully utilize these systems. 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 communication and data sharing behaviors. They provide a way to design and create robust and understandable parallel multicore applications rapidly. We use these high level parallel pattern characteristics in developing synthetic multicore benchmarks.

Benchmarks represent software workloads for current and future multicore systems and they are used for early design exploration and to evaluate performance, power consumption, and reliability of new multicore systems. 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.

Existing benchmark suites such as PARSEC, Rodinia as well as the embedded multicore benchmark suite EEMBC MultiBench are big and rely on presence of shared memory architectures, or Pthreads, OpenMP, and OpenCL libraries as well as uniform CPU 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.

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 work, 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 [1], 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 are similar on average 92% to the original benchmarks. We also found that the synthetics that correctly captured the parallel patterns in the originals have a high level of similarity.

• *E. Deniz and A. Sen are with the Department of Computer Engineering, Bogazici University, Istanbul, Turkey 34342. E-mail: etem.deniz@boun.edu.tr, alper.sen@boun.edu.tr. Brian Kahne and Jim Holt are with Freescale Semiconductor Inc., Austin, TX, USA. E-mail: brian.kahne@freescale.com,jim.holt@freescale.com*

In particular, this paper 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.
- 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.
- We developed MINIME tool and experimentally validate our techniques on both x86 and Power Architecture systems using PARSEC, Rodinia and EEMBC Multibench 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.

2 Parallel Patterns

Architectural patterns are fundamental organizational descriptions of common top-level structures observed in a group of software systems [2]. 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 [3]. Similarly, a parallel pattern language which is a collection of design patterns, guiding the users through the decision process in building a system has been developed [4]. 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 paper, we use the term parallel pattern as synonym for parallel software architectural pattern.

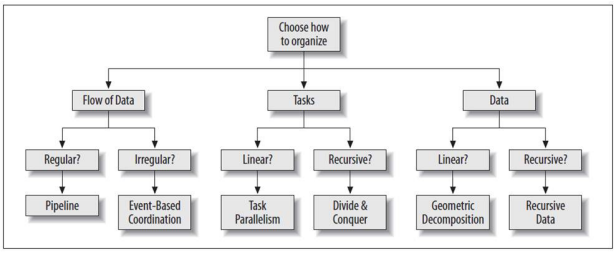

Fig. 1. Parallel Patterns for Software [5]

There exist three classes of parallel patterns based on organization of tasks, data, and flow of data. Figure 1 shows parallel patterns in a decision tree [5]. 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 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

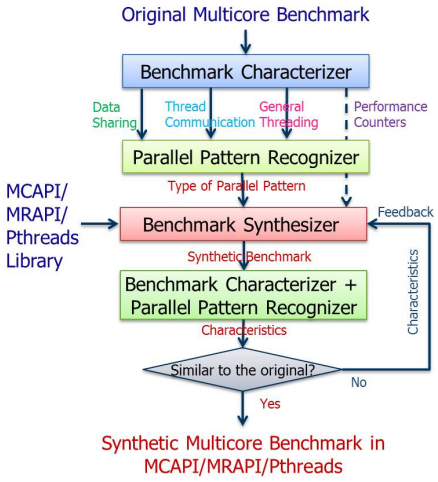

Fig. 2. MINIME: Pattern-Aware Multicore Benchmark Synthesizer Architecture

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.

Figure 2 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*. Next, we explain these modules in detail.

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 [6], [7] that use a high number of such characteristics. As can be seen in Table 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 [8]. 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.

TABLE 1
Multicore 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

Data Sharing Characteristics: Sub-characteristics in the data sharing group are *private, read-only, producer/consumer*, and *migratory*, similar to [8], [9]. We have formalized this concept in this work 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 technique uses binary instrumentation that models memory accesses per cacheline. If $\#readers = \#writers = 1$, we say that the data has *private* sub-characteristic. If $\#readers > 0$ and $\#writers = 0$, we say that 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. If we have $\#readers > \#writers$, then we have *producer/consumer* sub-characteristic. Otherwise, we have $\#readers \leq \#writers$ and *migratory* sub-characteristics, where a thread reads and writes to a shared data item and this behavior is repeated by many threads.

Thread Communication Characteristics: Sub-characteristics in the thread communication group are *none, few*, and *many*, similar to [8]. We have formalized this concept in this work 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 $numTH^2$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

When ($commTH < numTH$) and ($sharedCL \leq 0.3$), thread communication characteristic of an application is *none*. When ($commTH = numTH$) and ($0.3 \leq sharedCL \leq 0.8$), the thread communication characteristic is *few*. When ($commTH > numTH$) and ($0.8 \leq sharedCL \leq 1$), the thread communication characteristic is *many*. 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.

General Threading Characteristics: We keep track of the following general threading sub-characteristics for each thread. These are *program counter (PC)*, which allows us to determine whether the threads are executing the same task or not, *dynamic instruction count*, which allows us to determine whether the threads are balanced or unbalanced. We also keep track of the *creator* and *creation/exit time* of each thread. We calculate the *lifetime* of a thread by using the creation and exit times of the thread.

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

Characterization Tools: We use a dynamic binary instrumentation tool, named DynamoRIO [10], which is similar to Pin [11], 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 [12], 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. 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 [13] to obtain microarchitecture dependent characteristics.

4 Parallel Pattern Recognition

We use k-Nearest Neighbor (kNN) classification where k is 1 to decide the parallel pattern of an application. 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 but CCR given in Table 1 as the key characteristics. We do not use CCR because data sharing and thread communication characteristics implicitly cover this information. We developed our reference behaviors for each group of characteristics by investigating the behavior of threads for each pattern type in the literature [5]. We then validated that these characteristics and reference behaviors are indeed observed in multi-threaded applications for which we knew the pattern for. These applications were not used in the experiments. We describe the reference behaviors for each group of characteristics and for each parallel pattern in Table 2 [8].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 2 all sub-characteristics are most similar for threads in an application with GD pattern.

TABLE 2
Data Sharing [8], Thread Communication, and General Threading Reference Behaviors

Char.	Sub-char.	TP	DaC	GD	RD	Pl	EbC
Data Sharing	Private	****	*	*	****	*	*
	Read-only	***		*	***		
	Prod/Cons		**	****			
	Migratory	**	***	**		****	****
Thread Comm.	None	****	***			*	
	Few	*	**	*	*	****	****
	Many		*	****	****		
General Threading	PC	***	**	****	****		
	Dyn. Inst. Count	**	*	****	**	**	***
	Creator	****	**	****	***	***	*
	Creation Time	***	*	****	**	***	**
	Exit Time	**		***		**	*
	Lifetime	**	**	****	*	***	**

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 and the lowest score is 0. Hence, we calculate data sharing score, thread communication score, and general threading scores 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.

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

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.

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 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 5.4 and generate the synthetic given in Fig. 5. 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.

- $Step$ 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

TABLE 3
Algorithm for generating the code for a worker 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

- $Step$ 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 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.

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.

We use the error rate to quantify the similarity of a synthetic benchmark and an original application. Given a similarity metric, mt, and the value of mt for the synthetic and original application as $mtsyn$ and $mtorg$, respectively, we define the *error rate* for mt as, $errorrate_{mt} = |(mtsyn - mtorg)|/mtorg$. The *individual similarity scores* range from 0 to 100 and are calculated as $[1 - errorrate_{mt}] \times 100$. Finally, we calculate an *overall similarity score* (oss) as an equal weighted average of all of the similarity scores.

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 similarity scores and overall similarity score are satisfied or the user defined threshold for the number of iterations is reached.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers


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

Fig. 3. Code block to increment/decrement IPC, CMR, and BMR

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 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. Figure 3 shows the C code blocks we use to increment/decrement IPC, CMR, and BMR. 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.

5.4 A Detailed Example

We demonstrate our technique on a multi-threaded matrix multiplication application. Due to lack of space, we omit the original code and only show the synthetic in Fig. 5. 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

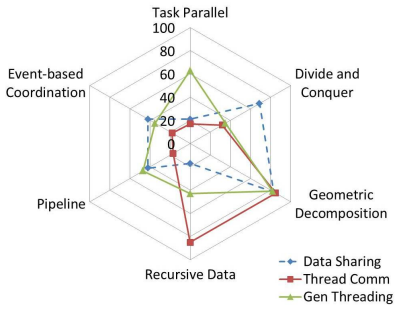

Fig. 4. Parallel Patterns Scores of Matrix Multiplication

```
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   /* code block for worker thread 2 */
16   /* code block for worker thread 3 */
17   return NULL;
18 }
19
20 int main(int argc, char **argv) {
21   /* initializations */
22   /* code block for computation */
23   /* create and run all the threads, then wait for the threads */
24   /* code blocks for CMR, IPC, and BMR to match similarity */
25   exit(0); /* global memory is removed automatically */ // Step 12
26 }
```

Fig. 5. Synthetic Matrix Multiplication Benchmark

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 4 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. 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 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 candi-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

TABLE 4
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

date synthetic benchmark has the following similarity scores: iss_{IPC} = 33, iss_{CMR} = 24, iss_{BMR} = 73, iss_{CCR} = 93, oss = 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 individual and overall similarity scores as follows: iss_{IPC} = 92, iss_{CMR} = 94, iss_{BMR} = 91, iss_{CCR} = 94, oss = 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 Fig. 5. The execution times of the original and synthetic are 0.08 and 0.02 seconds, respectively. Hence we have a 4x speedup. We achieve large speedups for large scale applications as will be shown in the experiments.

6 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 4. 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 [14], Rodinia (OpenMP) [15], and EEMBC MultiBench [16] 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. PARSEC is a well-known, open-source multi-threaded benchmark suite with fundamental parallelism constructs. Rodinia is a benchmark suite for heterogeneous computing and covers a wide range of parallel communication patterns, synchronization techniques, and power consumption. EEMBC MultiBench uses a thread-based API to establish a common programming model and targets the evaluation of scalable symmetrical multicore processor (SMP) architectures with shared memory. 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. 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. Due to lack of space, 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.

6.1 Evaluation of Benchmark Synthesis

Table 5 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. The column *Speedup(x)* shows speedup obtained in terms of execution time and the column *CodeSize(x)* 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

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

TABLE 5
Pattern Recognition and Synthesis Results

Original Benchmark				Synthetic Benchmark				
Suite	Benchmark	LOC	Parallel Pattern	LOC	#iter	Speedup(x)	CodeSize(x)	Overall Similarity (%)
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

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 21x (without `CFD Solver`) and the average code reduction is 14x. Note that *CodeSize(x)* 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 10174x 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 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 branch misprediction rate 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 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.

6.2 Assessing Similarity

We next compare the similarity of our synthetic benchmarks with the original benchmarks using similarity metrics described in Section 5. These metrics are IPC, CMR, BMR, and CCR. Due to lack of space, we do not show results for 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 error for each metric.

Figures 6, 7, and 8 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

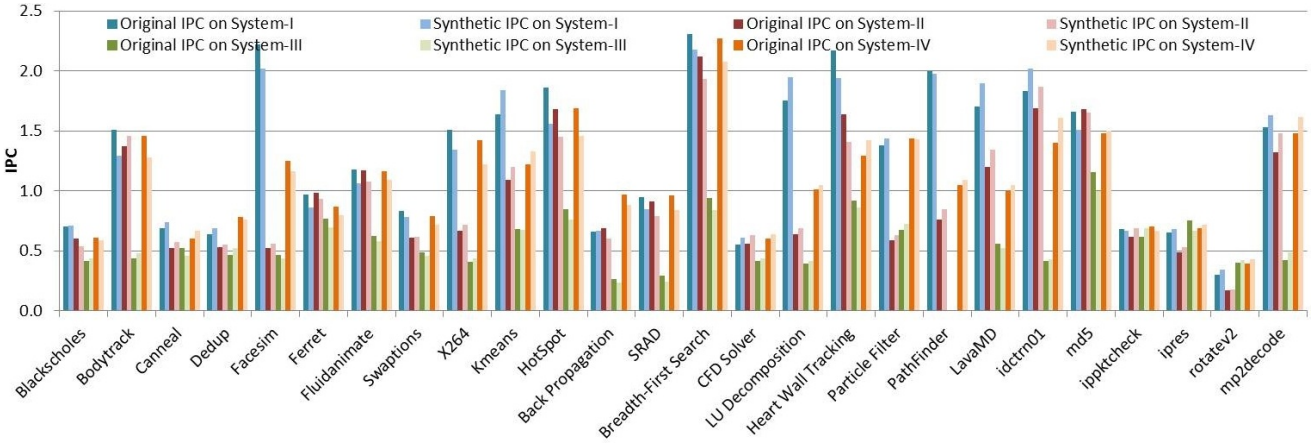

Fig. 6. Comparison of IPC between the synthetic and original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III.

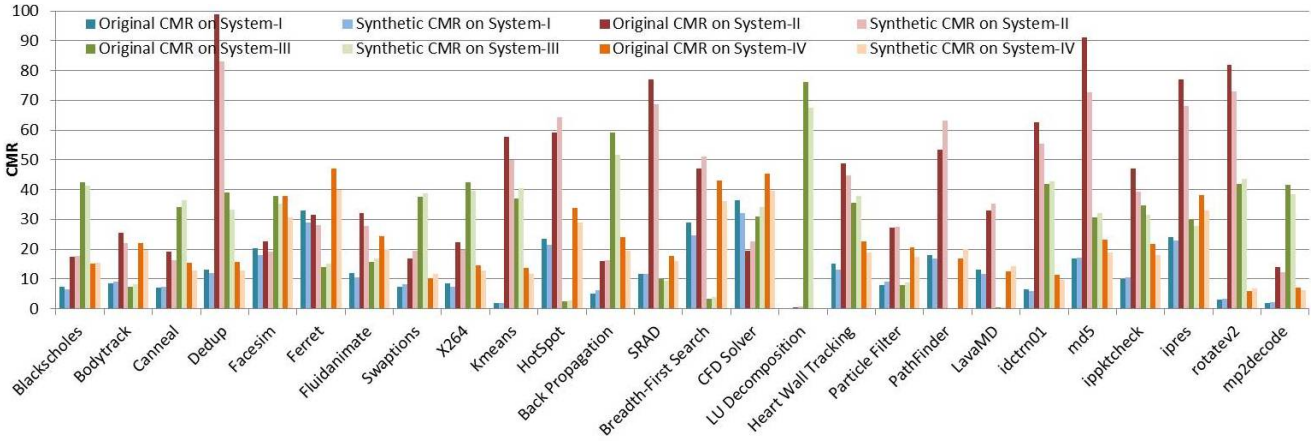

Fig. 7. Comparison of CMR between the synthetic and the original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III.

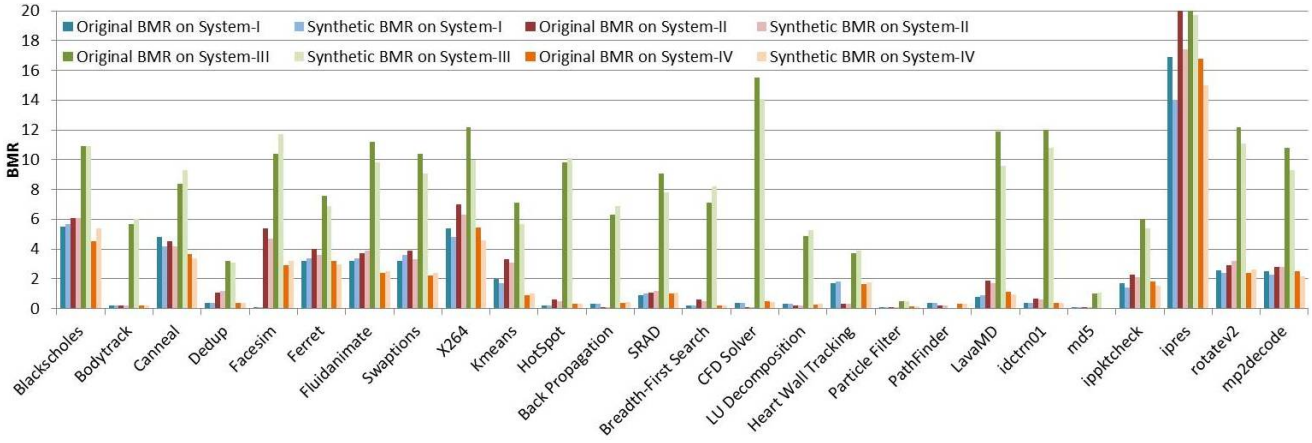

Fig. 8. Comparison of BMR between the synthetic and the original benchmarks. The synthetic generated on System-I is used on System-II and System-IV and re-synthesized for System-III.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

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, 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.

6.3 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 6 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%, 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.

6.4 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. Due to lack of space we display only the IPC values of original benchmarks for small, medium, and large inputs on System-I in Figure 9. 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. 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.

6.5 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 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 it has the correct parallel pattern, we also have the highest overall similarity score. For example, 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2349522, IEEE Transactions on Computers

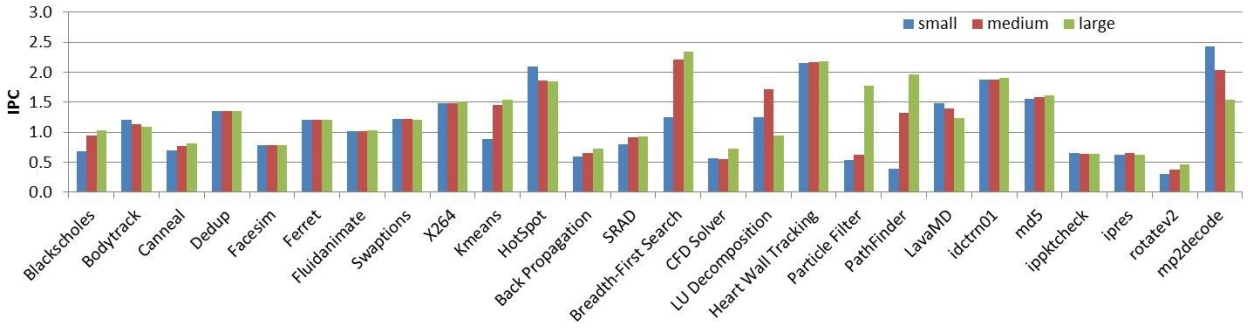

Fig. 9. IPC values of original benchmarks for small, medium, and large inputs on System-I.

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.

6.6 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 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. 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 6.4 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. 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 2x or decreased by more than 0.5x 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 (> 2x 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.

7 Related Work

Gamma et al. [3] introduced design patterns for object-oriented programming, similarly architectural patterns for parallel applications are given in [2]. Explicit knowledge of parallel pattern composition semantics has been exploited in previous work [17] to meet performance and power goals. Our parallel pattern recognition work is based on the work by Poovey et al. [8] where they implement pattern recognition in software. We have formalized this concept and presented new reference behaviors for thread communication and general threading. They detect parallel patterns from PARSEC and SPLASH-2 benchmarks. The accuracy of their pattern recognition technique is 50%, whereas we have a complete match on patterns in PARSEC. Also, in [18], the same authors use parallel patterns for dynamic thread mapping where they implement pattern recognition in hardware for speed purposes since their algorithm needs to work dynamically and periodically during the execution of the program. Since we only need to run pattern recognition once and not during the execution of a program, a software solution as in [8] suffices.

There has been prior work on characterizing benchmarks mentioned in this paper. In [14], 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. [15] 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 [19], 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. Poovey et al. [20] 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 [21], [22], [23]. These studies use microarchitecture dependent metrics like cycles per instruction, cache miss rate, and branch misprediction rate. In [24] and [25] the authors propose microarchitecture independent metrics for characterizing workloads.

Several works use statistical simulation for synthetic workload generation [26], [27], [28], [29]. However, in these works mainly single threaded applications and microarchitecture dependent characteristics have been used, whereas our work targets multi-threaded applications and uses microarchitecture independent characteristics.

So far, synthetic benchmarks have mainly been developed for sequential applications [6], [30]. There are some recent synthetics for multi-threaded applications in [23], [31], and [7] that target performance and power characteristics and may have lower error rates. These synthetics do not target embedded multicore systems, which we can, thanks to using MCA libraries. Also, their synthetics use low level assembly language, whereas we generate multi-threaded benchmarks as readable and portable C code. 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.

Portability of synthetics with changing microarchitectures has been studied in the literature [7], [24], [25]. In [32] and [33], 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 [34], 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 our work, 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. [35] 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.

In this work, we extended our preliminary work [36] in which we generated synthetic benchmarks in C using MCA libraries for embedded multicore systems. The improvements we made are described in the contributions section of the introduction.

Multicore Communications API (MCAPI) which is a lightweight message passing API and Multicore Resource API (MRAPI) which specifies essential application-level resource management capabilities are two of the standards developed by Multicore Association (MCA) [1], [37] 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.

8 Conclusions and Future Work

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. Ex-

periments show that our synthetic benchmarks are similar with the original benchmarks with respect to several metrics. They are also faster (on average 21x) and smaller (on average 14x) 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.

In the future, we plan to generate synthetic benchmarks that preserve power consumption characteristics. Also, we plan to detect composition of parallel patterns and changes of patterns over application phases.

ACKNOWLEDGMENTS

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

REFERENCES

[1] J. Holt, 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.

[2] J. L. Ortega-Arjona and G. Roberts, "Architectural Patterns for Parallel Programming," in *European Conference on Pattern Languages of Programs (EuroPLoP)*, 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994.

[4] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects," in *Workshop on Parallel Programming Patterns (ParaPLoP)*, 2010.

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

[6] A. Joshi, L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," in *IEEE International Symposium on Workload Characterization*, 2006.

[7] K. Ganesan and L. K. John, "Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 833–846, 2014.

[8] J. A. Poovey, B. P. Railing, and T. M. Conte, "Parallel Pattern Detection for Architectural Improvements," in *USENIX conference on Hot topic in parallelism (HotPar)*, 2011.

[9] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of Splash-2 and Parsec," in *IEEE International Symposium on Workload Characterization*, 2009.

[10] "DynamoRIO Dynamic Instrumentation Tool Platform, http://dynamorio.org/," 2013.

[11] C.-K. Luk, 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," in *ACM Conference on Programming Language Design and Implementation*, 2005.

[12] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and Scalable Memory Shadowing," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[13] "Linux profiling with performance counters, https://perf.wiki.kernel.org," 2013.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization*, 2009.

[16] "Embedded Microprocessor Benchmark Consortium, http://www.eembc.org," 2013.

[17] J. Holt and H. Hoffmann, "SEEC-AP: Self-Aware Software Architecture Patterns," in *International Workshop on Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE)*, 2012.

[18] J. A. Poovey, M. C. Railing, and T. M. Conte, "Pattern-Aware Dynamic Thread Mapping Mechanisms for Asymmetric Manycore Architectures," Georgia Institute of Technology, Tech. Rep., 2011.

[19] S. Che, 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," in *IEEE International Symposium on Workload Characterization*, 2010.

[20] J. A. Poovey, 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.

[21] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the Impact of Input Data Sets on Program Behavior and its Applications," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–33, 2 2003.

[22] S. Bird, A. Phansalkar, L. K. John, A. Mercas, and R. Idukuru, "Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture based processor," in *SPEC Benchmark Workshop*, Jan. 2007.

[23] K. Ganesan, L. K. John, V. Salapura, and J. C. Sexton, "A Performance Counter Based Workload Characterization on Blue Gene/P," in *International Conference on Parallel Processing (ICPP)*, 2008.

[24] K. Hoste and L. Eeckhout, "Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics," in *IEEE International Symposium on Workload Characterization*, 2006.

[25] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring Benchmark Similarity Using Inherent Program Characteristics," *IEEE Transactions on Computers*, vol. 55, pp. 769–782, 2006.

[26] M. Oskin, F. T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs," in *International Symposium on Computer Architecture*, 2000.

[27] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," in *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[28] L. Eeckhout, 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," in *International Symposium on Computer Architecture*, 2004.

[29] L. V. Ertvelde and L. Eeckhout, "Benchmark Synthesis for Architecture and Compiler Exploration," in *IEEE International Symposium on Workload Characterization*, 2010.

[30] A. Joshi, L. Eeckhout, and L. John, "The Return of Synthetic Benchmarks," in *SPEC Benchmark Workshop*, 2008.

[31] C. Hughes and T. Li, "Accelerating Multi-core Processor Design Space Evaluation using Automatic Multi-threaded Workload Synthesis," in *IEEE International Symposium on Workload Characterization*, 2008.

[32] J. Jo, L. K. John, M. Reese, and J. Holt, "Validation of Synthetic Benchmarks by Measurement," in *Workshop on Unique Chips and Systems (UCAS)*, 2010.

[33] L. K. John, J. Jo, , and K. Ganesan, "Workload Synthesis for a Communications SoC," in *Workshop on SoC Architecture, Accelerators and Workloads*, 2011.

[34] V. Deshpande, X. Wu, and F. Mueller, "Auto-generation of Communication Benchmark Traces," *SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 99–105, 2012.

[35] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: Effective Selection of Data Structures," in *ACM Conference on Programming Language Design and Implementation*, 2011.

[36] E. Deniz, A. Sen, J. Holt, and B. Kahne, "Using Software Architectural Patterns for Synthetic Embedded Multicore Benchmark Development," in *IEEE International Symposium on Workload Characterization*, 2012.

[37] "Multicore Association, http://www.multicore-association.org," 2013.