Using Software Architectural Patterns for Synthetic Embedded Multicore Benchmark Development

Etem Deniz*, Alper Sen*, Jim Holt† ‡, Brian Kahne†
*Department of Computer Engineering, Bogazici University, Istanbul, Turkey 34342
†Freescale Semiconductor Inc., Networking Systems Division
‡MIT Computer Science & Artificial Intelligence Laboratory
etem.deniz@boun.edu.tr, alper.sen@boun.edu.tr, {jim.holt@freescale.com,jholt@csail.mit.edu}, brian.kahne@freescale.com

***Abstract*—Benchmarks capture the essence of many important real-world applications and allow performance, and power analysis while developing new systems. Synthetic benchmarks are a miniaturized form of benchmarks that allow high simulation speeds and act as proxies to proprietary applications. Software architecture principles guide the development of new applications and benchmarks. We leverage software architectural patterns in developing synthetic benchmarks for embedded multicore systems. We developed an automated framework complete with characterization and synthesis components and performed experiments on PARSEC and Rodinia benchmarks. Our benchmarks can be run on any given infrastructure, that is, SMP or message passing, unlike previously developed benchmarks. Hence, this allows us to target heterogeneous embedded multicore systems. Our results show that the synthetic benchmarks and the real applications are similar with respect to various micro-architecture dependent as well as independent metrics.**

***Keywords*-multicore; workload characterization; software architectural pattern; synthetic benchmark; embedded multicore**

I. Introduction

As single core processors rapidly reach the physical limits of possible complexity and speed, multicore processors are growing as a new industry trend. Embedded multicore systems are being deployed in many domains ranging from medical to automotive to networks. However, designing multicore hardware is hard because the concurrent nature of multicores leads to complex behaviors that are difficult to analyze. Designing multicore systems is especially harder without workloads. Workloads can include real-world applications such as customer codes or benchmarks. Benchmarks are a good proxy for workloads that capture the essence of many important real-world applications. They are used to design and evaluate the performance of new computer systems. For example, PARSEC [1] targets multithreaded applications for shared memory architectures, Rodinia [2] targets heterogeneous systems including GPUs, EEMBC [3] contains applications for embedded systems, and NAS parallel benchmarks [4] contain scientific workflows for message passing architectures.

Workloads allow the designers to perform tasks such as early architectural exploration and predicting performance of new architectures. These tasks require high simulation speeds that may be difficult to accomplish when the workloads, either customer codes or benchmarks, are big. Furthermore, proprietary customer codes may not be available to the hardware designer. Hence, there is a need to develop high speed simulation benchmarks and benchmarks that can act as proxies for proprietary customer codes. Also, traditional benchmarks such as PARSEC, Rodinia as well as the embedded multicore benchmark suite EEMBC multibench all rely on presence of shared memory architectures, or Pthreads, OpenMP, OpenCL libraries as well as uniform CPU ISAs. Heterogeneous embedded multicore systems may not be able to use these benchmarks as they may not support such architectures or libraries. There is a need to develop workloads suitable for any given infrastructure, that is, SMP or message passing architectures, as well as workloads suitable for heterogeneous embedded multicore systems.

In order to address above problems with workloads, we develop synthetic benchmarks for embedded multicore systems suitable for any given infrastructure. Synthetic benchmarks do not perform any useful computation. However, the characteristics of real applications can be approximated by these small, simple and accurate programs. These benchmarks can be run on hardware simulation models, where the performance of the workload is crucial for timely development or on actual hardware, where the workload needs to capture more of the characteristics of the original application so that sensitivity analysis can be performed. Multicore Association (MCA) [5] provides a basic framework for developing applications for any given infrastructure on heterogeneous embedded multicore systems. We show that the synthetic benchmarks can run using MCA APIs (Multicore Communications API (MCAPI) or Multicore Resource API (MRAPI)), that provide a portability layer for heterogeneous systems. This allows us to demonstrate that the synthetics do not require SMP and shared memory, yet achieve similar characteristics to the original application.

In order to develop a synthetic benchmark, the first step is to characterize the given workload. Characterization consists of a description of the workload by means of quantitative

parameters and functions; the objective is to derive a model able to show, capture, and reproduce the behavior of the workload and its most important features. Workload characteristics can be divided into micro-architecture independent characteristics such as instruction mix, instruction level parallelism, data locality, thread communication; or micro-architecture dependent characteristics such as branch miss prediction and cache miss rate. In fact, significant work has been done to characterize single threaded benchmarks [6], [7]. Although, there has been work in multithreaded program characteristics such as memory level parallelism, ultimately the synthetic benchmark is a low level program. The development of synthetic benchmarks for multicore systems demands high level characteristics since our goal is to develop synthetic benchmarks suitable for any given infrastructure and low level characteristics simply do not allow the portability that we require.

In this work, we leverage software architecture patterns for capturing high level characteristics of multicore workloads. Such characteristics have not been previously used for synthetic benchmark development. Software architecture defines the components that make up a software system, the roles played by those components, and how they interact. A systematic way to describe software architectures is through patterns. A pattern is a high quality general solution to a frequently occurring problem. Patterns reduce the effort in developing new applications and programming paradigms and ease their adoption. Since we target multicore applications, we exploit parallel software architecture patterns as novel micro-architecture independent characteristics to develop synthetic benchmarks. Mattson et al. [8] describe parallel patterns such as pipeline, task parallel, divide and conquer and others in their work. These patterns have been extended in a pattern language with dwarfs such as dense linear algebra, dynamic programming, and structured grid in [9]. We show that patterns provide a sufficiently high-level framework for creating synthetic benchmarks, and there is simply a requirement for some basic infrastructure such as that provided by MCA.

Manually developing benchmarks for heterogeneous multicore embedded systems is both error prone and labor intensive. We develop an automated framework that can analyze a given workload and can create a synthetic benchmark for whatever infrastructure we have. We validate the applicability of software architectural patterns, in particular, parallel patterns, for developing synthetic multicore applications. We performed experiments on PARSEC and Rodinia benchmarks. Our results show that the synthetic benchmarks and the real applications are similar with respect to various micro-architecture dependent and independent metrics. We use metrics such as software architectural patterns, communication/computation as well as IPC, cache miss rate, and branch miss rate. Our benchmarks are similar up to 95% and on average above 86%. The readability of our benchmarks in C language is also much higher than previous synthetic benchmarks that generate assembly level benchmarks.

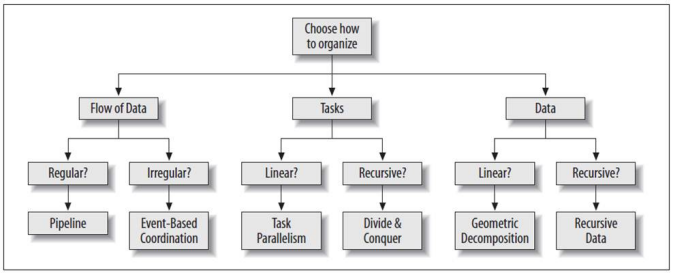


Figure 1: Parallel Patterns for Software [8]

We summarize our contributions as follows:

- We exploit software architectural patterns in characterizing and generating synthetic multicore applications.
- Our synthetic benchmarks can run any given infrastructure, that is, both SMP operating system or POSIX API as well as MCA APIs. Hence, it supports heterogeneous embedded multicores.
- We experimentally validate that our synthetic benchmarks achieve similar characteristics with the original workloads.

II. Software Architectural Patterns

Architectural patterns are fundamental organizational descriptions of common top-level structures observed in a group of software systems [10]. 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 understand complex software systems in larger conceptual blocks and their relations, thus reducing the adoption complexity.

Architectural design patterns have been developed for object-oriented software and have been found to be very useful [11]. 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 [12]. In a pattern language, patterns are organized into a hierarchical structure so that the user can design complex systems going though the collection of patterns. A parallel pattern language also provides domain-specific solutions to the application designers in less time.

Figure 1 shows parallel patterns in a decision tree. There exist three classes of parallel patterns based on organization of tasks, data, and flow of data. When a work is divided among several independent tasks, which can not be parallelized individually, the parallel pattern employed is task parallelism. In divide and conquer, a problem is structured to be solved in subproblems independently, and merging the outputs later. A data decomposition aligned with the set of tasks is designed to minimize communications between tasks and make concurrent updates to data safe. When the data

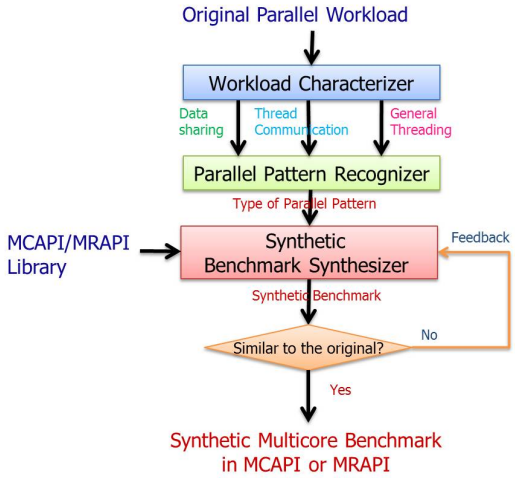


Figure 2: Synthetic Multicore Benchmark Framework

decomposition is linear, the parallel pattern that is employed is called geometric decomposition. Matrix multiplication, list and vector operations are examples of geometric decomposition. On the other hand, parallel pattern used with recursively defined data structures is called recursive data. Graph search, tree algorithms are example usages of recursive data. Apart form 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 subsequent computation, pipeline parallel pattern is used. Event-based coordination parallel pattern defines a set of tasks that run concurrently where each event triggers starting of a new task.

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 and we describe how we use them in the following sections.

III. Overall View of the Benchmark Framework

Figure 2 shows a high level view of the automated framework we developed. Our framework contains three main modules: *workload characterizer, parallel pattern recognizer*, and *synthetic benchmark synthesizer*. Workload characterizer obtains parallel workload characteristics by using a dynamic binary instrumentation tool. The parallel pattern recognizer decides the architectural parallel pattern from the workload characteristics. Finally, benchmark synthesizer generates a synthetic benchmark from the parallel pattern and the workload characteristics. Also, the specific MCA API is used during synthesis in order to generate benchmarks for a heterogeneous embedded system. Next, we explain each of these modules in detail.

IV. Parallel Workload Characterization

The goal of workload characterization is to produce some quantitative and qualitative relations and invariants that characterizes the behavior of a workload through experimental data. In this work, we are interested in those high level characteristics of a multicore application that allows us to determine its software architecture. All our characteristics are geared towards concurrent programs.

We analyze the workload characteristics in three groups: *data sharing, thread communication*, and *general threading*. Each group has sub characteristics. Sub-characteristics in the data sharing group are *read-only, migratory, shared*, and *private*. In read-only characteristics, the data is constantly being read by threads and is not updated. If the data is accessed by a single thread, we say that the data has private characteristics. In migratory characteristics, a thread reads and writes to a shared data item within a short period of time and this behavior is repeated by many threads. If multiple threads access the same data and at least one of the thread operations is write then this means that the data is shared between threads. Thread communication characteristics cover both the shared memory and message passing infrastructures. There exist three thread communication characteristics; namely, *none* (terminal thread or minimal communication), *few* (algorithmically defined), and *many* (data dependent) based on the amount of communication among threads. Furthermore, knowing the direction of communication is also valuable. Finally, we keep track of the following general threading characteristics. These are *the number of the threads over time, lifetime of threads, Program Counter (PC) uniqueness between threads* (this allows to determine whether the threads are executing the same function or not), *dynamic instruction counts per thread* (this allows us to determine whether the threads are balanced or unbalanced). Moreover, we build a relationship graph between threads where we keep the creator of each thread and the creation/exit times of the threads.

V. Parallel Pattern Recognition

Once we obtain the workload characteristics, we use them to decide the high level characteristics of the parallel workload, that is, the software architectural pattern or the parallel pattern. Our recognizer can detect different phases in a program as is the case for real programs and find multiple-patterns that follow one and other in a workload. In order to recognize parallel patterns described above, we develop a set of reference behaviors that capture the characteristics each parallel pattern exhibits. We then measure the Euclidean distance between the given characteristics of the workload and the characteristics of the reference behaviors to decide the parallel pattern of the workload. The parallel pattern nearest to the workload characteristics gives the parallel pattern of the workload.

We developed reference behaviors by investigating the behaviors in the literature [8], as well as our own experience with multicore applications.

In a task parallel or recursive data pattern, private and read-only data are common because each thread does its work locally without sharing data. In pipeline and event-based coordination patterns, data migrates between stages in which different threads run. The divide and conquer pattern has producer/consumer sharing and migration characteristics because the producer thread creates consumer threads by dividing tasks and the threads communicate with each other by migrating data. In geometric decomposition pattern, we have the producer/consumer sharing characteristics since data is shared between threads, particularly between threads with neighboring data.

In task parallel and divide and conquer patterns, there exist no or few inter-thread communication. This is because, each thread does its work by mainly using local data. The geometric decomposition and recursive data patterns have high degrees of inter-thread communication such as in graph traversal algorithms, where each thread shares its result with neighboring nodes. The pipeline and event-based coordination patterns have few and algorithmically defined communication between threads.

We use PC uniqueness between threads as general threading characteristics. For each thread we keep its PC, and then we check whether many threads have the same PC or each thread has a unique PC. If most threads have a unique PC, the parallel pattern is pipeline or event-based coordination since each thread does different work in these patterns. Whereas, PCs are shared between threads in other patterns. The distribution of the number of dynamic instruction counts of the threads also changes from pattern to pattern. For example, threads have similar number of dynamic instructions in geometric decomposition pattern as the subproblems have similar size. Whereas, in divide and conquer pattern, the number of dynamic instruction counts differs between threads because the size of the subproblems can be different. Similarly, the lifetimes of the threads have similar values for geometric decomposition pattern since the size of the data that the threads work on is divided equally. In divide and conquer pattern, threads have different lifetimes since the problems that each thread solves change. While all threads are created at the beginning in task parallel and geometric decomposition patterns, threads are created dynamically during the execution in divide and conquer pattern.

For example, the parallel pattern of a workload 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 program is task parallel. An example of geometric decomposition pattern can be a workload with many producer/consumer and few migratory data sharing characteristics, many data dependent inter-thread communication, and balanced threads that the share same PC and created at the same time by one thread.

VI. Synthetic Benchmark Synthesis

In this part, we use the workload characteristics and software architectural patterns to generate a synthetic benchmark. Synthetic benchmarks are synthesized as C programs with MRAPI or MCAPI library. Since C is a high level programming language, our synthetic benchmarks are portable and human-readable. There are many knobs to control the behavior of the synthesizer and it can also work as a standalone tool, where required parameters can be input manually rather than being automatically inferred through characterization.

Our synthetic benchmarks preserve the micro-architecture independent and dependent behaviors, hence they preserve both the software architectural semantics and the performance characteristics of the original workload. Note that synthetic benchmarks also preserve the number of the threads as well as their hierarchies. In order to achieve this, we measure the similarity between the original workload and the synthetic benchmark with respect to several similarity metrics. The goal of the synthesizer is to reach a user defined similarity score in an iterative and automated manner. The *similarity metrics* that we use are the Parallel Pattern type (PL), Thread Communication (TC) behavior, Communication/Computation Ratio (CCR), Instructions Per Cycle (IPC), Cache Miss Rate (CMR), and Branch Misprediction Rate (BMR). Many of the micro-architecture dependent metrics have previously been used to determine similarity but the software architectural patterns have not been used during synthesis.

We use the error rate to quantify the similarity of workloads. Given a similarity metric, mt, and the value of mt for the synthetic and the original workloads as $mtsyn$ and $mtorg$, respectively, we define the *error rate* for mt as, $errorrate_{mt} = (mtsyn - mtorg)/mtorg$. We now describe how we calculate the *similarity score*, $sscore_{mt}$, for each metric mt. The Parallel Pattern (PL) similarity score is calculated by comparing first whether the original and the synthetic have the same number of patterns, if not, a score of zero is generated. If they have the same number of patterns then we check the ratio of the number of matching pattern types in both workloads to the number of all pattern types in the original workload. Our synthesis flow makes sure that this score is always 100% for our synthetic benchmarks. The Thread Communication (TC) similarity score is calculated by comparing the communication behavior between pairs of threads in the original workload and pairs of threads in the synthetic workload. We calculate it as follows: $sscore_{TC} = (CC + NN)/(CC + CN + NC + NN)$. For a given pair of threads (assuming the number of threads is the same for both workloads), if the threads are communicating in both the original and the synthetic workloads, we increment the integer value CC. If the threads are not communicating in both the original and the synthetic workloads, we increment

the integer value NN. If the threads are communicating in the original but not in the synthetic workload, we increment the integer value CN, and similarly we increment NC. This score gives us an accurate number in terms of the communication behavior. The Communication to Computation (CCR) similarity score is the average of the error rate of communication between the original and the synthetic workloads and the error rate of computation between the original and the synthetic workloads. The similarity scores for the remaining Instructions Per Cycle (IPC), Cache Miss Rate (CMR), Branch Misprediction Rate (BMR) metrics are calculated as the inverse of the error rate formula given above. Finally, we calculate an *overall similarity score* as an equal weighted average of all of the similarity scores but the PL score. This is because we make sure that the synthetic and the original workloads have the same types of patterns.

Now, we describe the synthesis steps in more detail. The first step in the synthesis flow is to create a candidate synthetic benchmark using the parallel pattern and other workload characteristics. The candidate benchmark has the same number of threads as the original. We add communication operations among threads in an ordered way. These communication operations are either read/write (in case of MRAPI) or message send/receive operations (in case of MCAPI). We also decide on the type of messages and the number of operations in this step. The candidate benchmark exploits the reference behavior and the characteristics described above. For example, for task parallel pattern, in order to obtain private data sharing behavior, we create local private data and add read and write operations on this data. For read-only data sharing behavior, we create a global data that is read-only. We do not add any inter-thread communication since in task parallel every thread works on its unique task. In order to obtain PC uniqueness in general threading behavior, each thread executes a separate function. To obtain the same lifetimes, the amount of computation (sleep) is increased or decreased.

After the candidate benchmark is created, we check the overall similarity score. If it is above the overall similarity score or the total number of iterations both set by the user then we have the final synthetic benchmark. Otherwise, we iterate until we reach the threshold. During iteration steps, we check which similarity scores are lower than the threshold similarity score and improve those metrics. For example, in order to improve $sscore_{TC}$, we either add the missing communication between threads in the synthetic that exist in the original workload or remove the extra inter-thread communication that exists in synthetic but not in original workload. When $sscore_{IPC}$ is low, we either insert a C code block with high (integer addition) or low (division) IPC to the candidate synthetic benchmark. If $sscore_{CMR}$ is low, then we insert a new code block where the data that are already in cache are accessed many times. Otherwise, our new code block includes accesses to data that are not already cached. Similarly, we add code blocks for other metrics.

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

VII. Experiments

We performed experiments to analyze correlation of synthetic and real (original) benchmarks. In order to show that our approach works across different number of multicores and cache sizes, we targeted different core and cache platforms. The experiments were performed on two hardware configurations. HW1 uses an i7 processor with 4 cores and 6MB cache, and HW2 uses dual Xeon e5520 processors with 8 cores and 8MB cache.

We used PARSEC [1], and Rodinia (OpenMP) [2] benchmarks as real benchmarks and generated synthetic benchmarks in MRAPI and MCAPI from them. PARSEC is a well-known, open-source multithreaded 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 use a dynamic binary instrumentation tool, named DynamoRIO [13] for gathering characteristics during the execution of a workload. We also use Umbra [14], which is an efficient and scalable memory shadowing tool built on top of DynamoRIO. We developed our characterizers as clients of DynamoRIO and Umbra. We also used perf tool [15] to obtain micro-architecture dependent characteristics. Our framework consists of nearly 6000 lines of C code. We ran the original and the synthetic benchmarks 10 times in order to obtain similarity scores. We also set the number of iterations to 20, the overall similarity score to 80% and individual similarity scores to 70%. We used the medium input for PARSEC and default input for Rodinia. All of our benchmarks can be downloaded from our website [1].

Table I shows the results of our pattern recognizer as well as the pattern of the benchmark known from the literature on PARSEC benchmark suite. We obtained 100% pattern match on all benchmarks. Note that this may take a few iterations. Our pattern recognizer was also able to recognize different patterns in a given benchmark such as task parallel and pipeline patterns in X264, although this is not explicitly stated in the literature. The correct patterns are crucial because recognizing a wrong pattern can result in wrong communication and computation behaviors in the

[1] http://www.cmpe.boun.edu.tr/~sen/iiswc2012

Table I: Pattern Recognition Results for PARSEC benchmarks

Original			Synthetic		
Benchmark	**LC**	**Parallel Pattern**	**LC**	**Parallel Pattern**	**#Iterations**
Blackscholes	1262	Task Parallel	116	Task Parallel	1
Bodytrack	7696	Geometric Decomposition	1197	Geometric Decomposition	5
Canneal	2794	Task Parallel	116	Task Parallel	1
Dedup	7125	Pipeline	756	Pipeline	1
Facesim	20275	Task Parallel	190	Task Parallel	1
Ferret	10765	Pipeline	2722	Pipeline	4
Fluidanimate	2784	Geometric Decomposition	867	Geometric Decomposition	9
Swaptions	1095	Task Parallel	189	Task Parallel	6
X264	38546	Pipeline	1647	Task Parallel + Pipeline	17

synthetic benchmark. This can also result in higher number of iterations to match the synthetic benchmark 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 benchmark with only 50% overall similarity score even after 20 iterations. However, our pattern recognizer recognizes the parallel pattern as pipeline and our synthesizer generates a synthetic benchmark in 4 iterations with 82% similarity. This observation explicitly indicates a relationship between the high level architectural pattern and other metrics given above. We also show the lines of code (LC) for the original and the synthetic benchmarks as well as the number of iterations it takes to generate the synthetic benchmark. It can be seen that the synthetic is much smaller and less complex than the original as expected, hence leading to high simulation speeds. Also, in general, we generate the synthetic after only a few iterations. X264 took 17 iterations because the synthetic benchmark is large in terms of lines of code as well as the number of library function calls. These result in high influence on the metrics that we are trying to match. Furthermore, X264 has two patterns that makes it harder to synthesize.

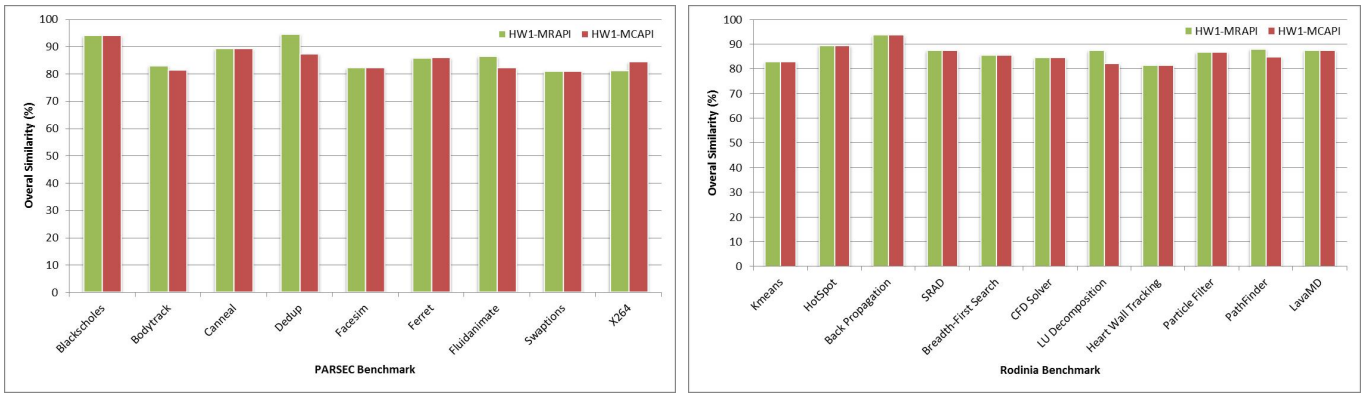
Note that parallel patterns of Rodinia benchmarks are not known from the literature, hence we did not display these results. However, our framework finds that Rodinia benchmarks have only task parallel and geometric decomposition patterns. This is expected because OpenMP does not support other patterns. That is, if the data used in OpenMP is private, then it results in task parallel pattern, otherwise the pattern is geometric decomposition. We observe that PARSEC and Rodinia benchmark suites do not contain all parallel patterns such as recursive data pattern. Also due to compilation and binary instrumentation problems we do not list results for all applications in these benchmark suites.

We next compared the similarity of our synthetic benchmarks with the real benchmarks using both micro-architecture independent metrics such as Parallel Pattern type (PL), Thread Communication behavior (TC), Communication/Computation Ratio (CCR) as well as micro-architecture dependent metrics such as Instructions Per Cycle (IPC), Cache (L1 and L2) Miss Rate (CMR), and Branch Misprediction Rate (BMR). We calculated the error between the synthetic benchmark and the original benchmark with respect to each of these metrics. We also present the average error for each metric. The first set of experiments are performed on hardware configuration HW1.

Figure 3 compares the overall similarity score of the synthetic benchmarks for MRAPI and MCAPI on HW1. The average similarity score is 87% and the minimum similarity score is 81% for MRAPI in Swaptions, x264, and Heart Wall Tracking. The maximum similarity score for MRAPI is Dedup with 95%. The average similarity score is 86% and the minimum similarity score is 81% for MCAPI in Swaptions, Bodytrack, and Heart Wall Tracking. The maximum similarity score for MCAPI is 94% for Blacksholes and Back Propagation. We observe that the synthetic and the original workloads are similar to each other over 80%, which was what was set by the user. These scores also show the high quality of synthetics.

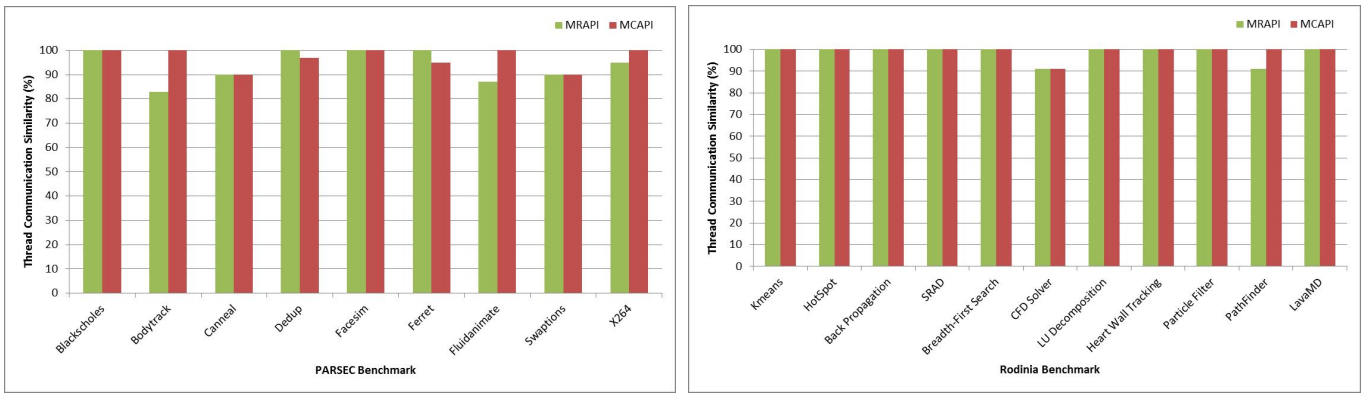
Figure 4 compares Thread Communication score of the synthetic benchmarks for MRAPI and MCAPI. The average error is 4% and the maximum error is 17% for MRAPI in Bodytrack. The average error is 2% and the maximum error is 10% for MCAPI in Canneal. MRAPI and MCAPI overall similarity and thread communication scores are close to each other. They both use the same library platform hence this is expected. Due to lack of space, unless specified otherwise, we display results for MRAPI synthetic benchmarks.

Figure 5 compares Communication/Computation between the synthetic and the real benchmarks for MRAPI. The average error is 19% and the maximum error is 29% for Kmeans. We observe that improving one metric can worsen others. Specifically, for Kmeans, we added a C code block in order to decrease the cache miss rate, which led to an increase in Communication/Computation error. Figure 6 compares IPC between the synthetic and the real benchmarks. The average error is 16% and the maximum error is 30% for Particle Filter. Figure 7 compares Cache Miss Rate between the synthetic and the real benchmarks. The average error is 16% and the maximum error is 30% for Ferret, and LU Decomposition. The reason why these synthetic benchmarks have large error is that the real benchmarks have the smallest (0.2%) and the highest (33.7%) cache miss rates that result in



(a) (b)

Figure 3: Overall Similarity Scores of synthetic benchmarks from (a) PARSEC, (b) Rodinia, for MRAPI/MCAPI on HW1



(a) (b)

Figure 4: Thread Communication scores of the synthetic benchmarks from (a) PARSEC, (b) Rodinia, for MRAPI and MCAPI

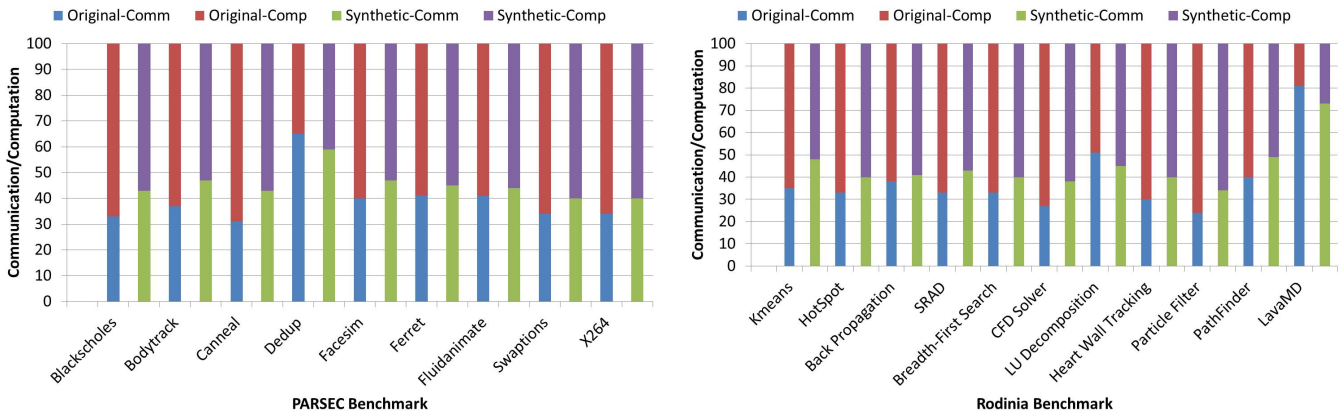
high loop counts with side effects in our synthetics. Figure 8 compares Branch Misprediction Rate between the synthetic and the real benchmarks. The average error is 12% and the maximum error is 29% for X264. Note that the average error for the above set of metrics is 16% and the maximum error is 30%. This is expected since our goal is to maximize those high level metrics such as the parallel pattern and thread communication. Even though these error results may seem high the overall score is still above 85% on average.

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

We next compare hardware configuration independence of our results by running experiments on HW2. All of the runs on HW2 use the same synthetic benchmarks synthesized from the HW1 configuration, not re-synthesized benchmarks. Figure 9 compares overall similarity scores of the synthetic benchmarks for MRAPI and MCAPI on HW2. The average similarity score is 85% and the minimum similarity score is 81% for MRAPI in Swaptions, X264, and Heart Wall Tracking. The maximum similarity score for MRAPI is 93% for Dedup and Back Propagation. The average similarity score is 84% and the minimum similarity score is 81% for MCAPI in Swaptions and Heart Wall Tracking. The maximum similarity score for MCAPI is 93% for Back Propagation. Figure 10 compares IPC between the synthetic and the real benchmarks on HW2. The average error rate is 16% and the maximum error rate is 30% for Particle Filter. We observe from HW2 results that the both the overall similarity scores and IPC scores are independent of hardware configurations. In other words, we observe nearly the same scores on both hardware configurations.

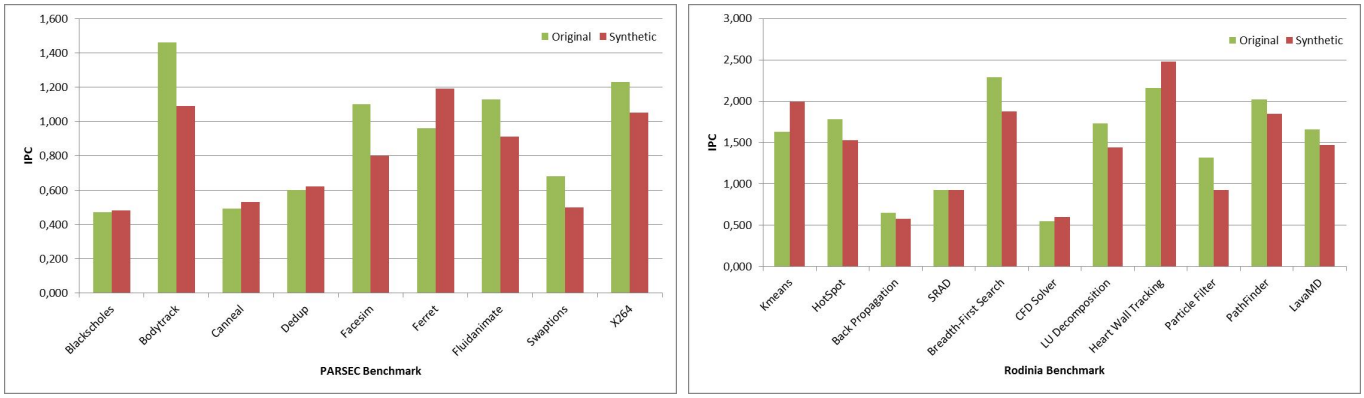
VIII. Related Work

There has been prior work on characterizing PARSEC. In [1], the authors analyze several characteristics such as



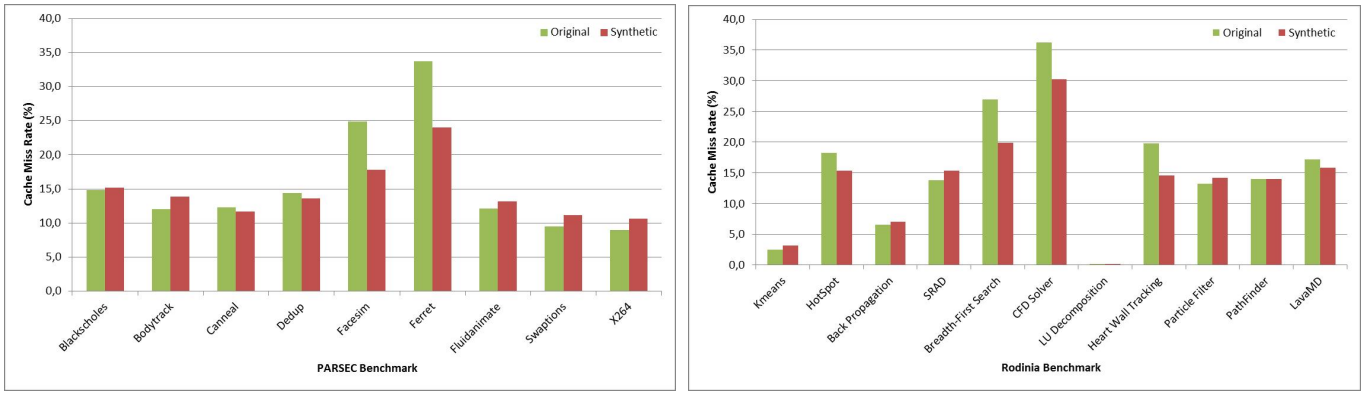
(a) (b)

Figure 5: Comparison of CCR between the synthetic and the original benchmarks from (a) PARSEC, (b) Rodinia



(a) (b)

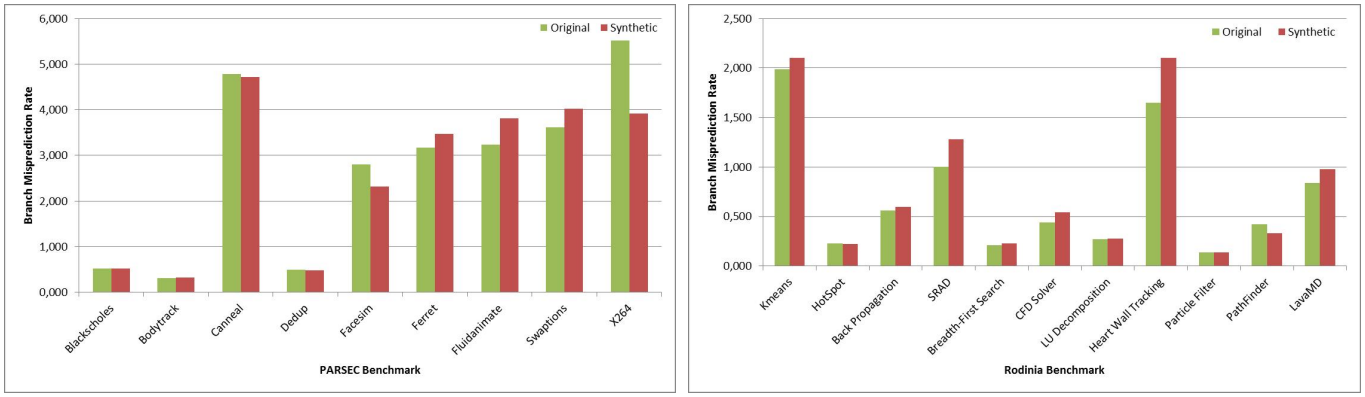
Figure 6: Comparison of IPC between the synthetic and the original benchmarks from (a) PARSEC, (b) Rodinia, for MRAPI



(a) (b)

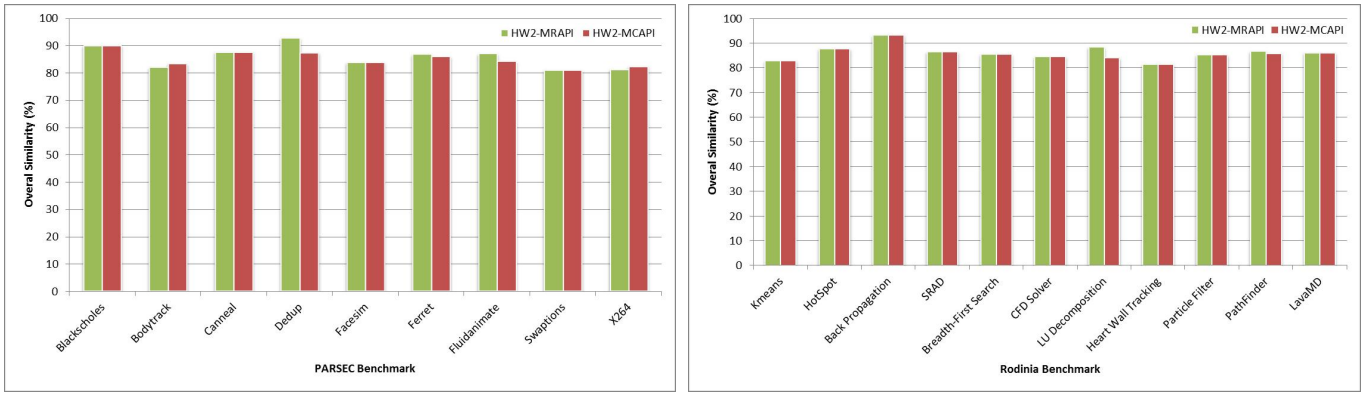
Figure 7: Comparison of CMR between the synthetic and the original benchmarks from (a) PARSEC, (b) Rodinia

data locality, effects of different cache block size, degree of parallelization and temporal and spatial behavior of communication. A comparison of PARSEC and Rodinia benchmark suites is given in [16], using instruction mix, working set, and sharing behavior characteristics. Hillenbrand et al. [17] present an architecture independent methodology for analyzing communication of multithreaded applications. The communication patterns they use are read-only, read/write,



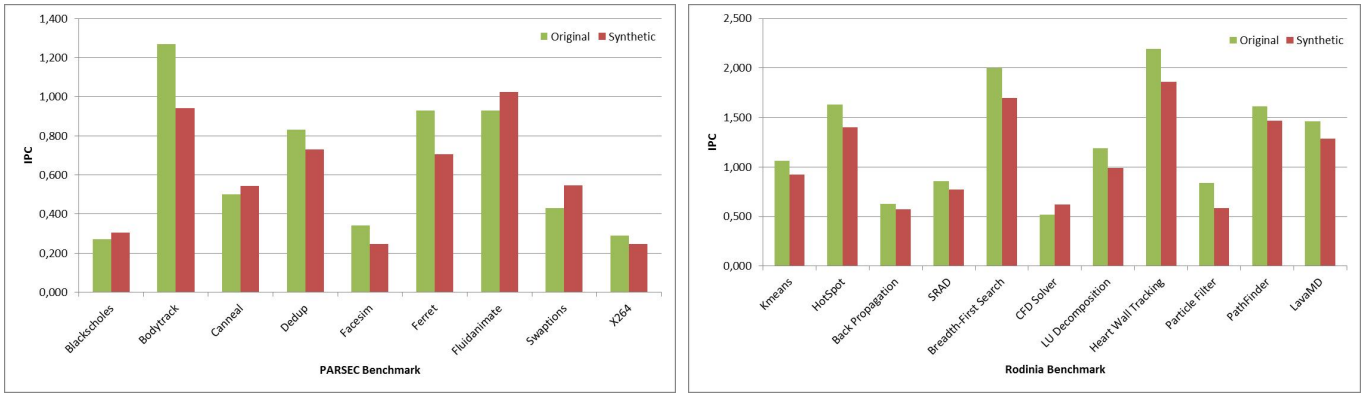
(a) (b)

Figure 8: Comparison of BMR between the synthetic and the original benchmarks from (a) PARSEC, (b) Rodinia



(a) (b)

Figure 9: Overall Similarity Scores of synthetic benchmarks from (a) PARSEC, (b) Rodinia, for MRAPI/MCAPI on HW2



(a) (b)

Figure 10: Comparison of IPC between the synthetic and the original benchmarks from (a) PARSEC, (b) Rodinia on HW2

producer/consumer and migratory. Bharathi et al. [18] characterized workflows from different scientific communities.

Benchmark synthesis [19] for performance evaluation has been previously investigated. Miniature clones of the applications at assembly level have been developed to minimize running time of the applications [20], [21], whereas we generate benchmarks as readable C code. So far, synthetic benchmarks have mainly been developed for sequential applications [6]. Although there are some recent multithreaded applications in [22], [23], these do not target embedded

systems. Also, the number of the metrics we used is much smaller, hence we can converge faster. In these works, micro-architecture independent characteristics have also been used for characterization but these do not include software architectural patterns.

Software architecture patterns [24] have long been studied. Gamma et al. [11] introduced design patterns for object-oriented programming. Poovey et al. [25] detect parallel patterns from PARSEC and SPLASH-2. The accuracy of their technique is 50%, whereas we have a complete match on patterns in PARSEC. Also, they do not use patterns in synthesis. Architectural patterns for parallel programs are given in [10]. Architectural patterns are classified based on functional parallelism, domain parallelism, or activity parallelism. The abstraction of the high-level architectural-skeletons as re-usable components for parallel computational patterns is given in [26]. Explicit knowledge of parallel pattern composition semantics has been exploited in previous work [27] to meet performance and power goals with dynamically self-optimizing parallel programs. Whereas, we detect sequential composition of parallel patterns and plan to detect concurrent composition in the future.

Multicore Association (MCA) [5] is a standard organization aiming to develop multicore software standards for heterogeneous embedded multicore systems. Multicore Communications API (MCAPI) and Multicore Resource API (MRAPI) are two of the standards developed by MCA. MRAPI is an API that specifies essential application-level resource management capabilities needed to coordinate concurrent access to multicore system resources. MRAPI standard handles memory management and supplies synchronization. MCAPI is a lightweight message passing API that aims to supply communication and synchronization between closely distributed embedded systems. MCAPI standard provides high performance, small memory footprint and scalable message-passing capabilities.

IX. Conclusions and Future Work

We developed an automated framework capable of generating infrastructure independent multicore synthetic benchmarks. We exploit software architectural patterns in characterizing multicore applications. These high level characteristics are essential in capturing the behavior of multicore applications. Our framework supports any given infrastructure, that is, both SMP operating system or POSIX API as well as MCA APIs. Hence, they can run on heterogeneous embedded multicores. Furthermore, our synthetics are in C and are readable. We experimentally validate that our synthetic benchmarks achieve similar characteristics with the original workloads. In the future, we plan to generate synthetics from other benchmarks such as NAS as well as real case studies. Also, we want to expand the hardware configurations that we experiment on and improve similarity scores for low level metrics as well.

Acknowledgment

This work was supported in part by Semiconductor Research Corporation under task 2082.001, Marie Curie European Reintegration Grant within the 7th European Community Framework Programme, Bogazici University Research Fund 5483, and the Turkish Academy of Sciences.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

[2] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[3] "Embedded Microprocessor Benchmark Consortium, http://www.eembc.org," 2012.

[4] D. H. Bailey, "Nas parallel benchmarks," in *Encyclopedia of Parallel Computing*. NASA Ames Research Center, 2011, pp. 1254–1259.

[5] "Multicore Association, http://www.multicore-association.org," 2012.

[6] A. Joshi, L. Eeckhout, and L. John, "The return of synthetic benchmarks," in *Proceedings of the 2008 SPEC Benchmark Workshop*, San Francisco, CA, USA, 1 2008, pp. 1–11.

[7] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2006, pp. 83–92.

[8] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley, 2005.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[10] J. L. Ortega-Arjona and G. Roberts, "Architectural patterns for parallel programming," in *European Conference on Pattern Languages of Programs (EuroPLoP)*, 1998, pp. 225–260.

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

[12] 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 *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10)*. New York, NY, USA: ACM, 2010, pp. 9:1–9:8.

[13] "DynamoRIO Dynamic Instrumentation Tool Platform, http://dynamorio.org/," 2012.

[14] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *The International Symposium on Code Generation and Optimization (CGO)*, Toronto, Canada, Apr 2010.

[15] "Linux profiling with performance counters, https://perf.wiki.kernel.org," 2012.

[16] 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 *Proceedings of the 2010 IEEE International Symposium on Workload Characterization (IISWC)*, 2010, pp. 1–11.

[17] D. Hillenbrand, J. Tao, and M. Balzer, "Alps: A methodology for application-level communication characterization of parsec 2.1," *Procedia Computer Science*, vol. 4, pp. 2086–2095, 2011.

[18] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS '08)*, nov. 2008, pp. 1 –10.

[19] C.-T. Hsieh and M. Pedram, "Microprocessor power estimation using profile-driven program synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 17, no. 11, pp. 1080–1089, 1998.

[20] K. Ganesan, J. Jo, and L. K. John, "Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010, pp. 33–44.

[21] A. Joshi, L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC)*, 2006, pp. 105–115.

[22] 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, pp. 330–337.

[23] K. Ganesan and L. K. John, "Maximum multicore power (mampo): an automatic multithreaded synthetic power virus generation framework for multicore systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. New York, NY, USA: ACM, 2011, pp. 53:1–53:12.

[24] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience (SPE)*, vol. 40, no. 12, pp. 1135–1160, 2010.

[25] J. A. Poovey, B. P. Railing, and T. M. Conte, "Parallel pattern detection for architectural improvements," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar '11)*, Berkeley, CA, USA, 2011.

[26] D. Goswami, A. Singh, and B. R. Preiss, "Architectural skeletons: The re-usable building-blocks for parallel applications," in *The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 1999, pp. 1250–1256.

[27] 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.