MINIME-Validator: Validating Hardware with Synthetic Parallel Testcases

Alper Sen
Department of Computer Engineering
Bogazici University, Istanbul, Turkey
Email:alper.sen@boun.edu.tr

Etem Deniz
Department of Computer Engineering
Bogazici University, Istanbul, Turkey
Email:etem.deniz@boun.edu.tr

Brian Kahne
NXP Semiconductors
Austin, TX, USA
Email:brian.kahne@nxp.com

***Abstract*—Programming of multicore architectures with large number of cores is a huge burden on the programmer. Parallel patterns ease this burden by presenting the developer with a set of predefined programming patterns that implement best practices in parallel programming. Since the behavior of patterns is well-known and understood they can also lower the burden for verification. In this work, we present a toolset, MINIME-Validator, for generating synthetic parallel testcases from a newly defined Parallel Pattern Markup Language (PPML) that uses the concept of parallel patterns. Our testcases mimic the behavior of real customer applications while being much smaller and can be used to generate traffic and validate e.g. inter-processor communication architectures. Experiments show that synthetic testcases can be used for finding representative hardware communication problems. To the best of our knowledge, this is the first time synthetic testcases using parallel programming patterns are used for hardware validation.**

I. Introduction

Embedded multicore architectures are gaining popularity in electronics. These types of hardware are used for many application domains including healthcare, automotive, and consumer electronics. Multicore hardware demands software that can exploit it for performance gains. High performance is obtained by the execution of multithreaded software where multiple operations can be completed simultaneously. However, concurrency brings the challenge of non-determinism that causes communication problems that may not be present in single core or sequential applications.

One solution to ease the burden of multithreaded programming is to use parallel programming patterns. Patterns bring best practices to commonly occurring programming challenges. Parallel patterns are high level characteristics that define the structure of a concurrent application in terms of communication and data sharing behaviors. They provide a way to design and create robust and understandable parallel applications rapidly. Pipeline and task parallelism are two examples of parallel patterns. Parallel patterns have been successfully used in many contexts including synthetic benchmark generation, dynamic task mapping and compiler optimization, parallel application development, and selecting an optimal architecture (see [1] for details). An example parallel pattern is

Multicore hardware validation is often handled by randomly generated testcases. This is useful for initial verification effort, but might not necessarily mirror real-world patterns of communication traffic. Unfortunately, real applications are too large to run via simulation and sometimes even too large for emulation. Hence these applications are typically run in post-silicon validation.

We propose a technique that allows for the generation of concise real world testcases to stress a system's inter-processor communication architecture, while limiting the total verification path length. Our test cases use parallel patterns that are observed in realistic applications. For example, a compression application such as dedup from PARSEC benchmark suite observes the pipeline pattern. In dedup application, there are five pipeline stages where one or more tasks independently work on a stage. The stages are reading inputs and generating coarse-grained chunks, anchoring each chunk into fine-grained segments, computing a hash value for each segment, compressing each segment, and assembling the deduplicated output stream. Thus, our technique can be used as an extra level of verification to catch more complex hardware bugs which might be missed via simpler random tests. Another benefit of our approach is that testcases can be run in pre-silicon during full-system simulation, which is generally not possible with real applications.

Given a multicore application, parallel patterns can be used to verify that the hardware is behaving according to the programmer's intention. For example, for an application with pipeline parallel pattern behavior, information is communicated (via the underlying architecture) from one pipeline stage (which may correspond to a core) to the next and the correct operation of the underlying network architecture can be accomplished by checking whether the correct information has arrived at the last stage of the pipeline. In this work, our goal is to exploit parallel patterns to validate the hardware synchronization mechanisms such as coherency, reservations etc. between the processor cores of embedded multicore hardware that runs concurrent applications.

We implement our techniques in a tool called MINIME-Validator, which is based on the synthetic application generator MINIME [2]. MINIME allowed the generation of synthetic applications from existing applications such as PARSEC suite by exploiting their parallel pattern. In this work, we extend MINIME such that synthetic applications (testcases) that use various parallel patterns can be *automatically* generated from user given Parallel Pattern Markup Language (PPML) specifi-

978-3-9815370-8-6/17/$31.00

cations rather than from existing applications, where PPML is a language that we define in this work. Another novelty of our approach is that each generated synthetic testcase also carries validation code appropriate for the type of parallel pattern used in the testcase. We experimentally validate our technique using Graphite hardware simulator [3] on a multicore architecture. We observe that the pattern of the testcase is related to the type of bug found.

II. Related Work

Since we generate synthetic testcases that can also be used as a benchmark suite we investigate such work in the literature. In the literature, there are synthetic benchmarks that are similar to the sequential [4], [5] or multithreaded applications [6], [7] that they are derived from. Synthetic benchmarks obtained from multithreaded benchmark suites such as PARSEC, Rodinia, and EEMBC that leverage parallel patterns have also been developed [2].

Parallel patterns have been used for deciding the type of multicore architecture. For instance, in the ParaPhrase project [8], [9], the authors decide the required resources in heterogeneous systems using the parallel pattern of a target application. In general, heterogeneous multicore architectures including larger and smaller CPU cores are suitable for divide and conquer and recursive data patterns. This is because threads in these patterns are unbalanced and assigning larger threads to the larger cores and smaller threads to the smaller cores provides load balancing. On the other hand, homogeneous multicore architectures are suitable for geometric decomposition pattern, which has balanced threads.

Although there are various works on validating parallel programs, that is not our goal in this paper. Our goal is in validating hardware architecture itself.

Testcases can be generated as user-directed, coverage driven [10], or randomly [11]. Most realistic testcases are used for post-silicon validation as they are large and slow [12]. Genetic algorithms have also been used for test case generation and validation of coherency protocols [13]. We generate fast and small synthetic tetscases based on PPML inputs that are representative of realistic applications thanks to using parallel patterns.

In this work, we extended MINIME tool [2] and obtained MINIME-Validator that generates verifiable and standalone synthetic testcases for multicore systems. Unlike MINIME, we do not need an actual application to generate the synthetic, the synthetic can be generated standalone. For this purpose, we defined an input specification language, PPML, to specify the attributes (properties) of the synthetic testcase. Then we generate characteristics from PPML that MINIME can use to generate synthetic testcases. We also add validation codes for each pattern type in the synthetic testcases expanding their benefits.

III. Parallel Programming Patterns

Parallel programming can be made attractive to general-purpose professional programmers by providing them with a

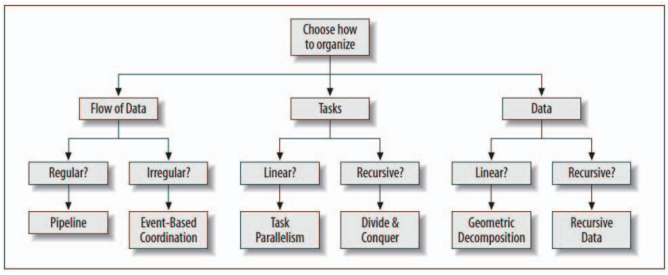

Fig. 1. Hierarchy of Parallel Programming Patterns [14]

set of patterns. Patterns are typically organized into a hierarchical structure so that the user can design complex systems going through the collection of patterns. Parallel patterns also provide domain-specific solutions to the application designers in less time. A widely-known set of parallel patterns has been proposed in [14]. In this set, 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 [14]. We briefly explain these patterns here. Further details can be found in [14], [2], [1].

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 decomposition is 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, 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 examples 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 the start of a new task. In this pattern, the interaction can take place at irregular and unpredictable intervals.

It has been shown earlier [2], [1] that multithreaded programs using above parallel patterns are characterized using

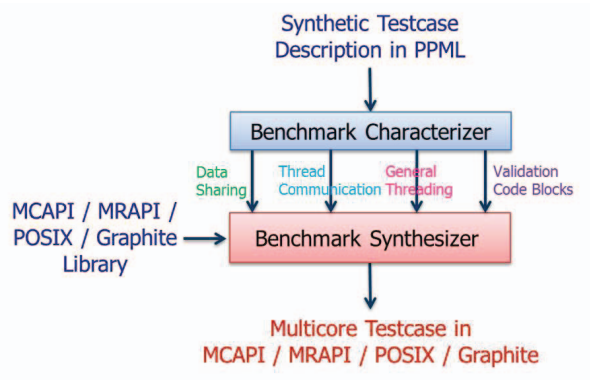

Fig. 2. MINIME-Validator: The tool takes PPML as input and outputs a multithreaded testcase complete with validation code

three classes of characteristics; data sharing (private, shared), thread communication (none, few, many), and general threading (thread id, creator thread, work size, etc.). Hence these characteristics capture the parallel pattern type of a program.

IV. Our Solution

Our MINIME-Validator tool can generate a synthetic benchmark (testcase) starting from a specification instead of a real application as was done in MINIME. The new tool is shown in Figure 2 with characterizer and synthesizer components. The characterizer component turns specifications given in Parallel Pattern Markup Language (PPML) into characteristics such as data sharing, thread communication, and general threading as used in MINIME as well as it generates validation code blocks. Then we use the synthesizer block to generate a synthetic testcase with the generated characteristics and validation codes. The synthesized testcase not only has the parallel pattern type specified in PPML specification but it also has code for validating that particular pattern. The testcases can use any of the listed library types hence they can be run on any architecture that supports these libraries. Our testcases are synthetic in the sense that they do not perform any useful functional computation but rather they implement the characteristics specified in the input PPML including the pattern type and thread behaviors.

A. Parallel Pattern Markup Language (PPML)

We defined a new language called Parallel Pattern Markup Language (PPML) to specify the characteristics and the parallel pattern of a testcase. We show the grammar of the language in Figure 3. PPML uses a style similar to XML where each line in a PPML specification defines the attributes of threads in a multicore testcase. We list some of these attributes in Table I, a complete list can be found on our website [].

As described in previous section a parallel pattern is specified by three set of characteristics; data sharing (private, shared), thread communication (none, few, many), and general threading (thread id, creator thread, work size). The attributes specified in Table I, allow us to capture these characteristics. We now explain some of these attributes. Each testcase has a pattern type. Each thread in a testcase has an associated thread id, work size denotes the amount of computation done by the thread where the user can increase or decrease this amount (which is captured as a parameter to a function), private data size denotes the size of thread local data, shared data size denotes the size of shared data among threads, and function is used for validating the pattern. Finally, PPML allows to specify patterns in a hierarchical manner, for example, in a pipeline pattern example, each stage in the pipeline can be executing an application with different pattern types.

Figure 4 displays an example PPML for a multithreaded application that uses pipeline parallel pattern. In this example, we have a 6 stage pipeline, where worker threads execute a recursive function or generates thread id (TID), and the main thread (id 0) executes no operation.

B. Validating Parallel Patterns

In this section we describe how our synthetic parallel pattern testcases can be used for validation. The validation code is automatically generated based on the pattern type, the number of threads, and thread attributes. For each pattern type, worker threads execute a well-known algorithm of that pattern type, then the main thread executes the same algorithm itself and compares its result with the result received from the collective execution of worker threads. We now describe in detail validation related activities done for each parallel pattern type.

Task Parallelism (TP): The main thread assigns each worker thread an arbitrary function for execution. Every worker thread sends its result to the main thread, which knows what the result should be, and verifies that the correct results are received. A simple worker thread function is to do some arbitrary arithmetic operation or just return thread id.

Divide and Conquer (DaC): Merge sort algorithm is executed by worker threads which communicate among each other using messages. Then the main thread executes merge sort itself and verifies that the results are consistent.

Geometric Decomposition (GD): This is similar to task parallel except the threads use the shared memory rather than messages for communication.

Recursive Data (RD): Depth-First-Search algorithm is executed on a data structure by worker threads that communicate among each other using messages. Then the main thread also searches the data structure itself and verifies that the results are consistent.

Pipeline (PL): Every worker threads sends the result of its execution of an arbitrary function (that is known by the main thread) to the thread in the next stage together with the results from previous stages. Finally all results are sent to the main thread. Since the main thread is aware of each function it then verifies that the correct results are received.

Event-based Coordination (EbC): This is similar to pipeline except that the data flow is irregular in this case. That is, messages are sent to threads in arbitrary stages.

Each pattern type can be used in different validation scenarios. For example, a pipeline pattern can be used to stress

```
program ::= ''<ppml_start>'' pattern ''<ppml_end>''
pattern ::= ''<pattern_start>'' ''<pattern:'' ptype ''>'' threads+ ''<pattern_end>''
ptype ::= ''task_parallel''|''divie_and_conquer''|''geometric_decomposition''|''recursive_data''|
        ''pipeline''|''event_based_coordination''
threads ::= ''<'' attr+ ''>'' | pattern
attr ::= ''tid: '' <integer> | ''stage: '' <integer> | ''wsize: '' <integer> | ''pds: '' <integer> | ''sds: '' <integer>
| ''function:'' func
func ::= ''NOOP'' | ''TID'' | ''SORT'' | ''SEARCH'' | ''RECURSION''
```

Fig. 3. Parallel Pattern Markup Language (PPML) Grammar

TABLE I
PPML ATTRIBUTES

Attribute	Description
pattern	Parallel pattern type: task parallel, divide and conquer, geometric decomposition, recursive data, pipeline, event-based coordination
tid (thread id)	unique thread ID. 0 denotes the main thread, others are worker threads
stage	used in pipeline pattern, denotes the pipeline stage that where the thread runs
wsize (work size)	parameter for the function (func) that the thread executes. Note that wsize = 1 for function TID.
pds (private data size)	the size of the local integer array that thread writes/reads
sds (shared data size)	the size of the shared integer array that threads share. e.g. messages sent over shared data in Graphite, hence sds=1 for pipeline below.
lc (loopcount)	used in pipeline pattern, denotes the number of times that a thread reads data from previous stage, works on it, and writes to the next stage
function	the type of function executed by the thread, example functions: SORT, SEARCH, TID, NOOP, RECURSION

```
<ppml_start>
 <pattern_start>
 <pattern: pipeline>
  <tid:4,stage:1,wsize:1,lc:10,pds:10,sds:1,func:TID>
  <tid:8,stage:2,wsize:4,lc:10,pds:10,sds:1,func:RECURSION>
  <tid:12,stage:3,wsize:7,lc:10,pds:10,sds:1,func:RECURSION>
  <tid:16,stage:4,wsize:9,lc:10,pds:10,sds:1,func:RECURSION>
  <tid:20,stage:5,wsize:1,lc:10,pds:10,sds:1,func:TID>
  <tid:24,stage:6,wsize:13,lc:10,pds:10,sds:1,func:RECURSION>
  <tid:0,stage:7,wsize:0,lc:10,pds:1,sds:1,func:NOOP>
 <pattern_end>
<ppml_end>
```

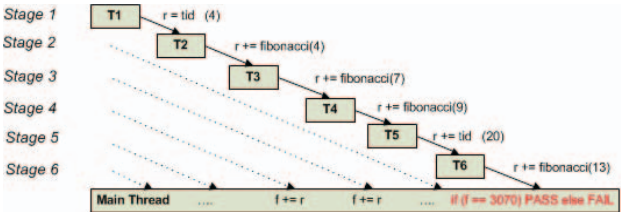

Fig. 4. Pipeline PPML Example and graphical representation of validation

data flow between cores, whereas a geometric decomposition pattern can be used for heavy data flow between two cores.

C. PPML Example

Once PPML specifications are converted into characteristics shown in the characterizer output of Figure 2, the results are passed to the synthesizer block to generate executable code that can use Pthreads, MCAPI/MRAPI, or Graphite communication libraries (based on user input). For each pattern type, there is a template parallel code that can be specialized based on each attribute in the PPML. We use the existing MINIME framework for synthesis. Additionally, we added template code for validation purposes as described above.

Figure 5 shows the synthetic testcase generated for the pipeline PPML specification given in Figure 4. Figure 4 also shows graphically the validation code for the example. We use the Graphite library in the test case. Specifically, there are 6 threads in the pipeline. Due to lack of space we only show thread 1, 2, and the main thread 0 (lines 1, 26, 57) in Figure 5. Every worker thread sends the result of its execution of either a recursive function such as fibonacci (line 43) or just its thread id (line 14) to the thread in the next stage together with the results from previous stages (lines 16, 39, 45). Finally, all results are received by the main thread (line 65). The main thread verifies that the correct results are received (lines 70-71). For example, if the payload from thread 2 (core 2) to thread 3 (core 3) were to be corrupted in the underlying communication architecture, this would be caught by the main thread as the expected result 3070 would not be received from thread 6.

V. EXPERIMENTS

We implemented our techniques in a new tool called MINIME-Validator that can be downloaded with our synthetic testcases from our website [15]. We use the Graphite architectural simulator to validate the effectiveness of synthetic testcases generated by MINIME-Validator. Graphite [3] is an open-source hardware architectural simulator that targets large-scale multicore processors with hundreds to thousands of cores. It provides both functional and performance modeling for cores, on-chip networks, and memory subsystems including cache hierarchies with full cache coherence. Graphite implements a layered communication stack. Application threads (cores) communicate with other threads (cores) via messages. Messages are routed and timed by target architecture network model. Many different architectures can be instantiated with Graphite. We assume that each thread runs on a different core.

We chose to focus on hardware communication problems as a case study although MINIME-Validator can be used for other hardware validation purposes as well. We developed a fault model for the communication mechanism in Graphite simulator. Our fault model targets real life communication problems, where messages between cores can be removed,

```
1   void *task1(void *param) {
2       /* initialize variables */
3       ...
4       CAPI_Initialize(1);
5       td->tid = 4;     //tid (thread id)
6
7       for (localMemIt = 0; localMemIt < 10; localMemIt++) {     //pds (private data size)
8           localMemTemp = 0; /* assign localMemTemp */
9           localAddr1[localMemIt] = localMemTemp; /* write local mem */
10      }
11
12      for (loopCount = 0; loopCount < 10; loopCount++) {     //loopcount = 10
13          /* Description: code block for work function */
14          localAddr1[0] += td->tid; //wsize=1 (work size), function = TID
15          //sds=1 (shared data size), stage=1 below
16          CAPI_message_send_w(me /* 1 */, 2, (char *)&localAddr1[0], sizeof(localAddr1[0]));
17
18          for (localMemIt = 0; localMemIt < 10; localMemIt++) {     //pds (private data size)
19              localMemTemp = localAddr1[localMemIt]; /* read local mem */
20              /* use localMemTemp */
21          }
22      }
23      return NULL;
24  }
25
26  void *task2(void *param) {
27      /* initialize variables */
28      ...
29      CAPI_Initialize(2);
30      td->tid = 8;
31
32      for (localMemIt = 0; localMemIt < 10; localMemIt++) {     //pds (private data size)
33          localMemTemp = 0; /* assign localMemTemp */
34          localAddr1[localMemIt] = localMemTemp; /* write local mem */
35      }
36
37      for (loopCount = 0; loopCount < 10; loopCount++) {        //loopcount = 10
38          // sds=1 (shared data size), stage=2 below
39          CAPI_message_receive_w(1, me /* 2 */, (char *)&message, sizeof(message));
40          localAddr1[loopCount] += message;     // vtype (validation type)
41
42          /* Description: code block for work function */
43          localAddr1[0] += fibonacci(4); //wsize = 4 (work size), function = RECURSIVE
44          // sds=1 (shared data size), stage=3 below
45          CAPI_message_send_w(me /* 2 */, 3, (char *)&localAddr1[0], sizeof(localAddr1[0]));
46
47          for (localMemIt = 0; localMemIt < 10; localMemIt++) {     //pds (private data size)
48              localMemTemp = localAddr1[localMemIt]; /* read local mem */
49              /* use localMemTemp */
50          }
51      }
52      return NULL;
53  }
54
55  ... // Code for task3, task4, task5, task6
56
57  int main(int argc, char **argv) {
58      CarbonStartSim(argc, argv);
59      CarbonEnableModels();
60      CAPI_Initialize(0);
61      td->tid = 0;     //tid (thread id)
62      ...
63      for (loopCount = 0; loopCount < 10; loopCount++) {      //loopcount = 10
64         //sds=1 (shared data size), stage=0 below
65         CAPI_message_receive_w(6, me /* 0 */, (char *)&message, sizeof(message));
66         td->workload += message;
67      }
68
69      /* Validation at the Main Thread */
70      if (td->workload == 3070) { printf("PASS, Result: %d\n", td->workload);}
71      else { printf("FAIL, Result: %d, Expected: %d\n", td->workload, 3070); }
72
73      CarbonStopSim();
74      return 0;
75  }
```

TABLE II
EXPERIMENTS

Synthetic Testcase	F1	F2	F3	F4	F5	F6
Task Parallel (TP1)	1	0	0	0	0	0
Divide-and-Conquer (DaC1)	0	1	0	0	0	0
Geometric Decomposition (GD1)	1	0	1	0	0	0
Recursive Data (RD1)	1	0	1	1	0	0
Pipeline (PL1)	0	0	0	0	1	0
Event-based-Cooridination (EbC1)	0	1	0	0	1	1

TABLE III
FAULT TYPES

Fault Type	Target	Description
FT1	TP	At iteration 1, the content of message from Thread N to 0 is modified/dropped.
FT2	DaC	At iteration 1, the content of message from Thread N to M is modified/dropped.
FT3	GD	At iteration 100, the content of message from Thread N to 0 is modified/dropped.
FT4	RD	At iteration 500, the content of message from Thread N to 0 is modified/dropped.
FT5	PL	At iteration 10, the content of message from Thread N to N+1 is modified/dropped.
FT6	EbC	At iteration 10, the content of message from Thread N to M is modified/dropped.

delayed or modified. We implemented the fault framework in Graphite by instrumentation using the GNU compiler.

We instantiated a 64-core Graphite configuration with a full-map directory based MSI coherency protocol, and a 2D mesh interconnection network. We generated six different synthetic testcases with MINIME-Validator as shown in Table II one for each pattern type. In Table II, we list the faults that were detected by the validation codes in each synthetic testcase. For example, fault F1 (where at iteration 1, message from core 2 to main thread 0 is dropped) is detected by TP1, GD1, and RD1 because there is message passing from thread N to 0 in all these 3 testcases. Fault F2 (where at iteration 1, message from core 2 to core 4 is dropped) is not detected by TP1, GD1, RD1, PL1 since there is no message passing from thread N to M (where $M \neq N+1$) in these testcases. Whereas, F2 is detected by DaC1 and EbC1 since such message passing exists. In the case of fault F3 (at iteration 100, message from core 3 to main thread 0 is modified), although there is communication from thread N to 0 the number of iterations is only 1 in TP hence the fault cannot be detected. The other faults can be explained in a similar way. From the above experiments, we observed that certain fault types are detected by each parallel pattern type and describe these faults in Table III. In a real hardware implementation, these faults would equate to bugs in the reservation or coherency logic, e.g. a dropped message equates to a bug in the reservation logic, while a modified data value equates to a coherency bug.

VI. CONCLUSIONS AND FUTURE WORK

We developed MINIME-Validator, a tool that can be used for automatically generating synthetic parallel testcases from specifications given in Parallel Pattern Markup Language (PPML). These synthetic testcases mimic the behavior of real life applications while being much smaller than them. Thereby, they can be run during presilicon validation for the full system, whereas most traditional benchmark suites are too large to run during simulation and do not specifically target parallel pattern behavior. We defined PPML as a language that allows to specify synthetic testcases using parallel programming patterns. We validated the effectiveness of MINIME-Validator for finding communication problems in a multicore hardware architecture modelled using Graphite simulator. We observed that the parallel pattern used in the testcase is related to the type of bug found.

In the future, we plan to experiment with different architectures and target specific coherency bugs as well as compare our results with that of existing test suites.

REFERENCES

[1] E. Deniz and A. Sen, "Using machine learning techniques to detect parallel patterns of multi-threaded applications," *International Journal of Parallel Programming*, vol. 44, no. 4, pp. 867–900, 2015.

[2] E. Deniz, A. Sen, B. Kahne, and J. Holt, "Minime: Pattern-aware multicore benchmark synthesizer," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2239–2252, Aug 2015.

[3] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[4] 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.

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

[6] 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.

[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] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonz?lez-V?lez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer, "The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems," in *Formal Methods for Components and Objects*. Springer Berlin Heidelberg, 2013, pp. 218–236.

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

[10] S. Yang, R. Wille, D. Groe, and R. Drechler, "Coverage-driven stimuli generation," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, Sept 2012, pp. 525–528.

[11] D. A. Wood, G. A. Gibson, and R. H. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design & Test of Computers*, vol. 7, no. 4, pp. 13–25, 1990.

[12] B. W. Mammo, V. Bertacco, A. DeOrio, and I. Wagner, "Post-silicon validation of multiprocessor memory consistency," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 1027–1037, June 2015.

[13] M. Elver and V. Nagarajan, "Mcversi: A test generation framework for fast memory consistency verification in simulation," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[14] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[15] "MINIME-Validator, http://depend.cmpe.boun.edu.tr/tools/minime-validator," 2016.