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

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 speci-
fied 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

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 exe-
cuted 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 exe-
cuted 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 thread 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 sce-
narios. For example, a pipeline pattern can be used to stress
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,
```c
void *task1(void *param) {
    /* initialize variables */
    CAPI.Initialize(1);
    tid->tid = 4; //tid (thread id)
    for (localMemIt = 0; localMemIt < 10; localMemIt++) { //pds (private data size)
        localMemTemp = 0; /* assign localMemTemp */
        localAddr[localMemIt] = localMemTemp; /* write local mem */
    }
    for (loopCount = 0; loopCount < 10; loopCount++) { //loopcount = 10
        //sds=1 (shared data size), stage=1 below
        CAPI.message_send_w((char *)localAddr[0] + tid - tid > 0, (char *)localAddr[0], sizeof(localAddr[0]));

        for (localMemIt = 0; localMemIt < 10; localMemIt++) { //pds (private data size)
            localMemTemp = localAddr[localMemIt]; /* read local mem */
            /* use localMemTemp */
        }
    }
    return NULL;
}

void *task2(void *param) {
    /* initialize variables */
    CAPI.Initialize(2);
    tid->tid = 8;
    for (localMemIt = 0; localMemIt < 10; localMemIt++) { //pds (private data size)
        localAddr[localMemIt] = localMemTemp; /* write local mem */
    }
    for (loopCount = 0; loopCount < 10; loopCount++) { //loopcount = 10
        //sds=1 (shared data size), stage=2 below
        CAPI.message_send_w((char *)localAddr[0], sizeof(message) ) ;

        localAddr[loopCount] += message; //vtype (validation type)
        "localAddr[0] += fibonacci(4); "
        "wsize = 4 (work size), function = RECURSIVE"
        "sds=1 (shared data size), stage=3 below"

        CAPI.message_send_w((char *)localAddr[0], sizeof(localAddr[0]));

        for (localMemIt = 0; localMemIt < 10; localMemIt++) { //pds (private data size)
            localMemTemp = localAddr[localMemIt]; /* read local mem */
            /* use localMemTemp */
        }
    }
    return NULL;
}

int main(int argc, char **argv) {
    CarbonStartSim(argc, argv);
    CarbonEnableModels();
    CAPI.Initialize(0);
    tid->tid = 0; //tid (thread id)
    ... // Code for task3, task4, task5, task6
    CarbonStartSim(argc, argv);
    CarbonEnableModels();
    CAPI.Initialize(0);
    tid->tid = 0; //tid (thread id)
    ... // Code for task3, task4, task5, task6
    CarbonStartSim(argc, argv);
    CarbonEnableModels();
    CAPI.Initialize(0);
    tid->tid = 0; //tid (thread id)
    ...
    CAPI.message_send_w((char *)message, sizeof(message));
    td->workload += message;

    /* Validation at the Main Thread */
    if (td->workload == 3070) { printf("PASS, Result: %d\n", td->workload); }
    else { printf("FAIL, Result: %d, Expected: %d\n", td->workload, 3070); }
    CarbonStopSim();
    return 0;
}
```
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