Concurrent Generation of Concurrent Programs for Post-Silicon Validation

Allon Adir, Amir Nahir, Avi Ziv
IBM Research - Haifa, Israel
Email: {adir, nahir, aziv}@il.ibm.com

***Abstract*—The continuing trend toward increased parallelism in processor design can be seen in both the growing number of processor cores per system and in on-core hardware mechanisms that assist parallelism, such as multi-threading and cache hierarchies. This complexity exacerbates the problem of ensuring the functional correctness of such hardware systems. The growing importance of post-silicon validation is leading to an emerging type of parallel application, namely the hardware exerciser. We describe a method for exercising parallel hardware by generating pseudo-random concurrent test programs. The test generation is carried out on the tested parallel platform and thus the generator itself is also a concurrent program. We describe the challenges associated with this technology and the approach used by the Threadmill hardware exerciser, a tool developed for the post-silicon validation of the IBM POWER7 processor.**

I. Introduction

Functional verification of hardware designs is a process that seeks to demonstrate the compliance of a design with its specification [1]. This process is widely acknowledged as the main bottleneck in the hardware design cycle [2]. To date, most of the verification of complex designs, such as high-end processors, is done using dynamic verification [1]. In dynamic verification, the *design under verification* (DUV) is simulated (or emulated) with given stimuli and its behavior is checked to ensure that it operates according to its specification. Dynamic verification is a highly automated process that uses the computing power of thousands of workstations. However, it still requires hundreds of person years to create verification environments, debug failures, etc. To support the automation requirements, verification environments contain sophisticated random stimuli generators that generate high-quality random stimuli according to user specifications, checking mechanisms that automatically check if the behavior of the DUV is complying with its specification, and examining coverage collection and analysis to ensure that verification is thorough.

But even with all this verification effort, it is virtually impossible to eliminate all bugs in the design before it tapes out. In fact, statistics show that close to 50% of chips require additional unplanned tape-outs because of functional bugs. Moreover, in many cases, project plans call for several planned tape-outs at intermediate stages of the project before the final release of the system. As a result, an implementation of the system on silicon running at real-time speed is available. This silicon is used, among other things, as an intermediate and final vehicle for functional validation of the system in what is known as post-silicon validation.

Despite the common goals of pre-silicon verification and post-silicon validation, the major differences between the platforms dictate differences in the respective implementation of verification solutions. One difference is that post-silicon platforms provide significantly higher execution speeds. This speed allows much faster execution of tests, but it limits the ability of the environment to perform "smart" time-consuming activities during the execution of tests, such as generating high-quality stimuli and checking the behavior of the DUV. Another difference between the platforms is the level of observability provided in the domains.

As a result, similar challenges have different solutions in the pre- and post-silicon domains. For example, sophisticated stimuli generators that are the backbone of pre-silicon verification [3] are often replaced with simpler *exercisers* [4]. Once loaded to a system, exercisers continuously generate, execute, and check the results of test cases. Their on-platform test generation engine must be fast and light, and therefore simple, compared to technologies used for pre-silicon verification. Threadmill [5] is a "bare-metal" exerciser (i.e., running directly on silicon without a supporting OS layer), with a simple and fast pseudo-random generation engine, designed to support a unified pre- and post-silicon verification methodology. Threadmill is directable, enabling fine user direction and control over the generated tests. It supports various checking methods and provides mechanisms that help analyze failures.

Modern high-end processor systems are highly parallel. For example, the IBM Power 795 system contains 32 POWER7 chips, each with eight processor cores, with each core running four threads in parallel, for a total of 1024 parallel threads. The multiprocessing architectures of such systems are very complex [6]. To simplify the programming model, these architectures contain strict ordering rules that affect all the threads in the system. One example of this type of rule is *coherency*, which requires all threads to observe the same order of writing to the same memory address. However, to enable performance optimizations, the architectures allow *weak ordering*, which enables processes to execute and observe memory accesses in any order, as long as some ordering rules are complied with. In addition to these architectural complexities, the micro-architecture of such systems contains many mechanisms, such as caches and store queues, which implement the architecture and optimize the performance of the system.

The complexity of the multiprocessing aspects of a system makes its verification one of the biggest challenges of

the already complex and challenging verification process. To demonstrate that multiprocessing mechanisms operate as intended, the verification team needs to generate interesting scenarios that hit corner cases in the verified system and check that the behavior of the system in those scenarios is correct. Both tasks are far from simple. Generating interesting scenarios requires good control of the generated instructions and the timing in which they are executed. In pre-silicon verification, this is done using the testing knowledge and sophisticated generation engine of the stimuli generator [7]. As far as checking is concerned, determining that the behavior of the DUV is correct (for example, that all threads execute and observe memory accesses in allowed orders) is an NP-complete problem [8]. Therefore, most verification environments rely on checking internal behaviors to ensure proper system behavior [9].

The challenges of multiprocessor verification are amplified in post-silicon validation, where sophisticated stimuli generators are replaced by exercisers and the limited observability does not enable internal behaviors to be checked to determine compliance with multiprocessor specifications. This paper presents some of the techniques used in Threadmill to address the multiprocessing verification challenges in a post-silicon exerciser. Threadmill's main challenge is to generate stimuli that reach all the corner cases of the multiprocessing features often enough to ensure that bugs in those features are exposed. Threadmill does so using three main features. First, the directability of the tool allows users to specify the skeleton of the required scenarios using Threadmill's template language. Second, Threadmill partitions the test memory to regions with specific roles and owners to allow all kinds of memory collisions. These collisions are required in multiprocessing verification, but Threadmill carefully controls the collisions, allowing the final results of the tests to be checked. Finally, to ensure that the timing conditions of corner cases are created, Threadmill uses *irritation threads* [10]. These threads repeat parts of a given scenario quickly and frequently, ensuring that various interactions between these parts and the rest of the scenario occur in many different ways, including those needed to create the interesting timing conditions.

In addition to the challenges and solutions above, Threadmill is executed on the target system itself, i.e., Threadmill and specifically its generation engine are distributed applications. This creates a challenge of generating tightly coordinated test programs in a distributed manner. The two simplest solutions, using a single thread for centralized generation or using multiple threads that synchronize whenever coordination is needed, are inefficient. Therefore, Threadmill uses a shared random seed to ensure that the same random decisions are reached whenever coordination is needed. This solution has two main requirements. First, all the threads must use shared random generation in the same way. In addition, the threads must use another private seed to allow different generated stimuli by the various threads. The shared and private seeds allow distributed coordinated generation with minimal synchronization between the threads (basically, just to distribute the shared seed).

Threadmill, and the approach advocated in this paper, focuses on functional verification at the processor or multiprocessor levels including the memory sub-system and the caches. Our work did *not* extend to non-functional aspects (such as power or performance verification) and to levels beyond the multiprocessor (such as IO or communication). Threadmill was used for the first time in the verification and bring-up of the IBM POWER7 processor [11], where it was used to help verify the multi-threaded and multiprocessor aspects of the POWER7 chip. Threadmill's ability to generate high-quality concurrent test programs provided high coverage of the multi-threading and multiprocessing features of the chip and helped reveal several difficult bugs.

The rest of the paper is organized as follows. In Section II, we provide some background on functional verification of hardware systems and discuss the differences between pre- and post-silicon verification techniques. In Section III, we present Threadmill, our post-silicon exerciser. Section IV describes the challenges of multiprocessor verification. Section V describes the concurrent test programs that Threadmill generates. In Section VI, we show how Threadmill is implemented as a concurrent program to take advantage of all the available threads during generation. Section VII shows our experience with Threadmill in the verification of the POWER7 processor. Section VIII concludes the paper.

II. Hardware Functional Verification

Functional verification of hardware seeks to demonstrate the compliance of a hardware implementation with its specification [1]. This process is performed as part of the hardware design cycle, starting from when the requirements of potential or actual customers are analyzed and a specification for the hardware architecture is compiled, to the verification of full physical systems that include the fabricated chips. To date, the leading techniques for functional verification are formal and dynamic verification. Formal verification techniques, such as model checking [12], "mathematically" prove that a given design behaves according to its specification. While formal verification techniques are very powerful, they are only relevant to small units due to their computational complexity. Therefore, dynamic verification is the main functional verification vehicle for large and complex designs.

Most of the verification effort is carried out before the hardware is fabricated in pre-silicon verification, but pre-silicon verification cannot find all the bugs in the design. Therefore, post-silicon validation also plays an important role in the overall verification effort.

A. Pre-Silicon Dynamic Verification

The verification process starts with a plan identifying areas in the DUV that need to be verified. For each such section of the DUV, the plan defines interesting scenarios and events that relate to it, the types of checking that need to be done to ensure its correct behavior, and measures for the thoroughness of the verification. These areas of the verification plan correspond to the three legs of dynamic verification, namely, stimuli

generation, checking, and coverage [1]. These elements are implemented in a verification environment that controls the DUV and observes it behavior.

In current practices, the three legs of dynamic verification are mostly automatic. Human effort in the process is invested in the creation and maintenance of the verification plan; design and implementation of the verification environment; analysis of coverage data; and (primarily) debugging.

The main vehicle for executing the DUV is software simulation. Software simulation is slow (6–9 orders of magnitude slower than silicon), but it provides high levels of controllability and observability of the DUV. These characteristics of software simulation dictate the requirements for the verification environment. The slowness of software simulation makes simulation cycles a precious commodity that must be used efficiently. Therefore, the generated stimuli must be of high quality, with a very good probability of generating an interesting test[1]. Similarly, checking needs to detect virtually any bug exposed by the stimuli, and to achieve this it can use the internal observability of the DUV. On the other hand, the slowness of the software simulator provides the verification environment with ample time to perform long and complex operations. The end result is complex, compute-intensive verification environments that provide high-quality services.

For example, pre-silicon stimuli generators must generate stimuli that are valid (i.e., with well-defined behavior in the DUV specification), interesting, and adhere to users' requests. In addition the generator must be able to generate many significantly different stimuli from the same test specification. State-of-the-art stimuli generators, such as IBMs Genesys-Pro [3], achieve this using several techniques. First, they provide their users with a test-template language [13] as an easy and accurate way for specifying the scenarios in the verification plan. The requirements from the generated stimuli are represented as a constraint satisfaction problem [14]. A random test is then created by a random sampling of the solution space of this constraint satisfaction problem. Embedding *testing knowledge* in the generator allows it to bias random decisions towards stimuli that cause interesting events [14].

B. Post-Silicon Validation

Once the silicon comes back from the fab, the long process of bring-up begins. One of the important roles of the bring-up process is continuing the functional verification of the chip on silicon. In this sense, post-silicon validation has common goals with pre-silicon functional verification, namely, both demonstrate the compliance of the DUV to its functional specification and strive to detect bugs left in the system while doing so. Post-silicon validation has many other goals, such as detecting electric and other non-functional problems and testing system software. Moreover, individual fabricated chips are tested for production flaws (this is termed manufacturing testing). These topics are beyond the scope of this paper.

[1]There are many factors that can make a test interesting. An explanation of those factors is beyond the scope of this paper.

In recent years we have seen a move toward a common methodology for pre- and post-silicon verification [5]. Despite the common goals and methodology, the differences between the pre- and post-silicon platforms dictate big differences in the tools and technologies used in these domains.

The main difference between the platforms is their execution speed. There is a difference of 6–9 orders of magnitude between the speed of software simulators and that of silicon. On silicon, verification tools cannot take advantage of the slowness of execution to perform complex computations between cycles. On the other hand, the high speed of silicon means that many more cycles are available, and therefore the average quality of verification per cycle can be much lower (as long as high enough peaks are maintained). As a result, post-silicon stimuli generators, for example, can be much simpler and faster than their pre-silicon counterparts.

A second difference is the long loading and initialization time of silicon. As a result, running short verification jobs on silicon is inefficient. A common solution to this problem is using solutions that load once and run "forever.

Another difference between the platforms is the level of observability they provide. Post-silicon platforms provide very limited observability. This means that pre-silicon checkers that rely on internal observations must be converted to internal checkers implemented on the silicon (a costly operation that is not always possible) or be abandoned (which requires other checking solutions).

The limited observability also affects the ability to debug the DUV and the software running on it. Therefore, post-silicon solutions tend to be simple. Another incentive for simplicity is the fact the the post-silicon tools run on the tested platform when it is presumed to contain bugs—especially during the earlier stages of the post-silicon validation effort. In the very first stages of validation, when the hardware is still unstable, only the simplest tools are deployed. Still, it occasionally happens that a bug is found during the operation of the tool rather than by a test that it generated. In addition, in these early stages the operating system (OS) and other supporting tools are not yet available. Post-silicon tools, therefore, often run "bare-metal", i.e., without an OS. This implies that they must supply the services normally provided by an OS (such as I/O and interrupt handlers) on their own. Later on, when the hardware becomes cleaner, the more sophisticated tools can be used.

Simplicity is also required to maintain as low a ratio as possible between time spent generating and checking a test case and the time spent running it. Reference models are a good example. Using a reference model on silicon could improve checking capacity, but the reference model itself is complicated software. Using it would significantly reduce the platform's utilization, making a reference model an inefficient choice for post-silicon validation.

In the next section, we show how these characteristics are considered when creating a stimuli generator for the post-silicon platform.

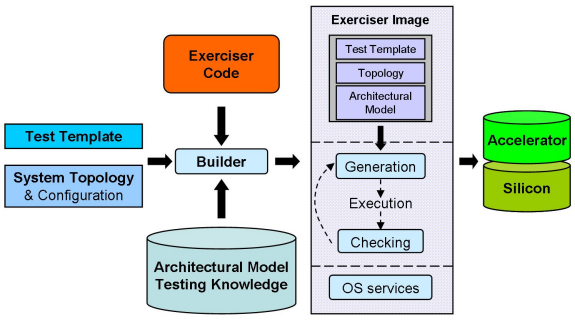

Fig. 1. Threadmill architecture

III. Threadmill

Post-silicon platforms provide very high execution speeds but incur relatively high costs in loading time and communication with the environment. As a result, post-silicon validation tends to rely more on longer running solutions, such as *post-silicon exercisers*, programs that run on the DUV and "exercise" it by testing interesting scenarios. An exerciser is a self-contained solution that performs a large number of tests after being loaded only once on the DUV, which makes it appropriate for post-silicon validation.

Threadmill [5] is a post-silicon exerciser developed as part of the post-silicon validation effort for IBM's POWER7 processor. It complies with the relevant requirements for such tools as described in Section II-B. Threadmill is an on-platform, "bare-metal" exerciser. Once it is loaded on silicon, it generates, executes, and checks tests indefinitely. Because of the requirement for simplicity of post-silicon solutions, Threadmill avoids sophisticated generation techniques often used by pre-silicon test generators, such as CSP (Constraint Satisfaction Problem [15]). For the same reason, the exerciser also does not rely on a reference model to perform the checking.

The high-level tool architecture of Threadmill is depicted in Figure 1. The main input to Threadmill is a test template that describes the required scenarios from the verification plan. Threadmill generates a large number of test programs that correspond to the user's test template, as described in Section V below. Other inputs include the architectural model, testing knowledge, and the system topology. The Threadmill execution process starts with a builder application that creates the executable *exerciser image*. The builder runs off-line, i.e., not on the tested system. The role of the builder is to convert the data incorporated in the test template and the architectural model into data structures; these structures are then embedded into the exerciser image. As opposed to the simplicity of the exerciser itself, the "off-line" operation of the builder can afford to spend time on sophisticated generation and checking, for example, by using a reference model.

The exerciser image is composed of three major components: a thin OS-like layer of basic services required for Threadmill's bare-metal execution; a representation of the test template, architectural model, and system configuration description as simple data structures; and fixed (test-template independent) code responsible for the exercising. The image created by the builder is then loaded onto the silicon platform where the exerciser indefinitely repeats the process of (a) generating a random test case based on the test template, the configuration, and the architectural model, (b) executing the test case, and (c) checking its results.

Execution of the same test case multiple times is used by Threadmill as a partial replacement for checking done by the reference model. This is done by observing certain resource values (such as registers and part of the memory) at the end of the test and checking their equivalence following different executions of the same test case. Running the same test case multiple times can result in different results even when bugs are not present. For example, when a test includes a write-write collision, the final value at the shared memory location depends on the order of the write operations. This requires that certain mechanisms be implemented in the generator to restrict the number of such unpredictable resources.

As mentioned earlier, Threadmill targets functional bugs—i.e., violations of the specified functionality of the processor. Its checking approach (multi-pass comparison) is less appropriate for detecting bugs in the data computation aspects but has proven to be effective for control-path oriented bugs, or bugs that reside at the intersection of the control- and data-paths. To increase the probability of exposing such bugs, it is beneficial to introduce some kind of variability into the different execution passes, while making sure that the variability still maintains the predictability of the compared resources. This can be done, for example, by changing the machine mode, or changing thread priorities[2].

The control-path required for multi-processor functions is fairly complex and induces many types of related bugs. These include, for example, deadlock or livelock situations, thread starvation and other effects of race conditions over shared memory. These types of bugs often invlove complex scenarios with delicate timing conditions. Threadmill is able to target such intricate scenarios with its rich test template language. It is able to cover many possible timings of the threads by the irritation method (described in the following sections) and by the mere weight of cycles that is available on the silicon platform.

IV. Verification of Parallel Processors

All of the high-end servers currently on the market are designed for parallelism. These systems often comprise multiple chips with several processor cores, each running multiple threads concurrently. Such systems support resource sharing, most commonly shared memory, between the threads. Threads can also share core-level resources such as special purpose registers. In this paper, we refer to all these systems as *MP* systems.

The continuing trend toward increased parallelism can be seen in the ever growing number of processors and threads

[2]Some variablity is automatically created, for example, due to the different state of the caches and the instruction scheduling in the different passes.

in a system. The latest IBM POWER processor system, IBM Power 795 [11], for example, can include 1024 threads running in parallel on 256 separate processor cores. To support high parallelism, the processor *micro-architecture* (i.e., the lower-level design implementation) includes sophisticated features that improve performance and facilitate the smoother operation of the concurrently running threads. The higher-level processor *architecture* (i.e., the specification of the processor as provided to the user, including its machine language and other structural and interface aspects) also includes MP-related sections. These sections include MP-related instructions, such as cache, semaphore-related instructions, and memory synchronizing instructions, which help order conflicting accesses by different processes to the same memory area.

Many MP architectures that support shared memory also define a shared memory model that relaxes the simple sequential consistency model [8] and allows some re-ordering of memory accesses to become visible to the programmer. This is done mainly to support performance boosting mechanisms such as caches, out-of-order, and speculative executions. Such *weaker* memory models [16], [17] still impose rules on the re- orderings that the processes may observe. For example, the PowerPC Sequential Execution Model rule [6], [18] allows re-orderings, provided that processes are not aware that their own operations have been re-ordered (while there are cases where a process may observe that operations of other processes have been re-ordered). The *Coherency* rule requires that no two processes are able to observe two accesses to the same memory location (or memory *granule*) in a conflicting order. The *Access Atomicity* rule states that an access composed of smaller accesses should appear to execute as a whole, without the interference of accesses made by other processes to the same resources. For example, the PowerPC guarantees atomicity for most accesses aligned to their size (e.g., a word aligned Store-Word access).

MP verification then addresses the task of ensuring the correct operation of the micro-architecture mechanisms that support the MP execution of the system and verifying that the design indeed conforms with the MP-related definitions and rules of the architecture specification. The sizes of today's MP systems imply a strict requirement for the scalability of the verification solution. Current software simulators cannot handle such large systems. That conclusion leads to increased reliance on post-silicon validation platforms where the actual system is targeted.

Other challenges of MP verification are connected with checking (i.e., identifying that a bug has occurred) and debugging (i.e., locating the root cause of the bug). The common checking method of comparing the program output to the expected valid output can be difficult to apply to MP tests, since these normally have many possible valid outcomes, depending on the relative timing of the processes. Debugging an MP test program suffers from the usual difficulties of debugging parallel applications, stemming from the asynchronous execution of the processes. A specific bug may occur based on very fine relative timing of the processes, which could be hard to define or reproduce. These difficulties are exacerbated when the checking and debugging are performed on post-silicon, since these platforms typically provide very limited observability and controllability over the state of the processor [5].

Scenarios that involve collisions, i.e., accesses to the same resources by different processes, trigger the more complex parts of the MP mechanisms [7]. Cases in which every process accesses an exclusive set of resources should be covered primarily during the uni-processor verification phase (where the processor is verified in an environment that doesn't include multiple processors). Some mechanisms, such as semaphores, are inherently intended for use in collision scenarios. In addition, most violations of the shared memory models rules (like coherency and access atomicity) can only be detected in scenarios with collisions. All this implies that generating scenarios that include collisions will increase the bug detection potential of the generated tests.

A collision naturally involves multiple processes that access the same resource. However, such *true sharing* can be difficult to check since it can multiply the number of possible valid outcomes of the scenario. For example, if two threads write different values concurrently on the same byte, then both values are valid outcomes for the byte. Keeping track of possible outcomes and avoiding an explosion in the number of different possible values (due to value propagation) can be difficult. One possible approximation of true sharing that does not suffer from these difficulties is when the colliding threads access the same cache line but in different locations. Known as *false sharing*, this is interesting for verification purposes since it triggers the same cache mechanisms.

Several types and aspects of collisions are worth verifying:

- Shared resources: Verification efforts should consider all the resources accessible by more than a single process. The most important case is shared memory, but chip-level registers can also be shared by several processors on the same chip. Multi-threading also allows several threads to use the same registers on a single processor. In addition, caches can be shared when several processes are allowed to simultaneously access the same cache, same cache row, or even the same cache line.
- Access orders: There are four possible *sense* orders for the colliding accesses: write-write, write-read, read-write, and read-read. Write-read and read-write access orders are distinct because they can trigger different coherency-related mechanisms. Collisions that include writes are more interesting for verification and pose the most difficult challenges for the test generator because they are the hardest to predict and check [7].
- Affinity: A collision can be characterized by the spatial *affinity* of the participating threads and collision resource locations. This refers to the amount of physical separation between the colliding threads, or between the threads and the collision resources. These could be located, for example, on the same core, the same processor, or on remote processor nodes in the system. Different affinity

levels typically trigger different design mechanisms that require dedicated verification.

The most interesting collisions occur when an access request arrives while another access to the same resource is being processed. Many design constructs are designed to address this situation; the validation process should test the constructs in a variety of timing situations (including "near cases" where the request almost arrives at or just misses the crucial timing). Some scenarios also call for the colliding accesses to be fully ordered (e.g., a read request to a cache line that was marked modified by a previous write).

Accesses do not necessarily have to initiate simultaneously to be close in time, but they do generally have to at least start within a short time frame. Threadmill synchronizes all the threads before they start to perform the test to increase the probability of real time contention. It also uses synchronization to achieve specific access ordering when desired. A proven technique for achieving high variability of collision timing on both data and computation resources is *thread irritation* [10]. According to this method, the multithreaded test case comprises a primary test case to be executed by a primary thread or threads and additional irritation test cases for one or more irritator threads. During execution, the irritators execute an infinite loop with a very small body comprising only one or two instructions. The irritation is continuously repeated as long as the primary threads are executing. This scheme guarantees that the irritation will occur in many different collision timings, relative to the primary scenario. Another positive aspect of irritation scenarios is that the irritator threads put stress on the execution units involved in the scenario, thereby reaching corner cases of these units' functionality.

When the primary test case finishes execution, it terminates all the irritator threads (for example, by signaling an asynchronous interrupt to the irritators). Normally, the irritator thread is not allowed to cause any unexpected exceptions or modify any memory used by the primary thread, thus enabling the primary scenario to execute as planned. However, the irritator can perform read-read actions, or false-sharing collisions with the primary thread. It can also compete for shared computation resources, such as an execution unit, shared by the primary and irritator threads. An actual design bug found in the early bring-up stages with this approach is described in Section VII.

V. Concurrent Test Programs

Threadmill was developed to enable a verification methodology driven by a test plan. The test plan includes a list of abstract coverage goals directed at specific design features to be tested and functional behavior to be verified. These coverage goals are translated into a collection of *test templates* that specify the desired test programs in abstract terms and serve as the primary input to Threadmill. The test templates are written in a manner that enables easy and accurate specification of the scenarios from the verification plan. Figure 2 displays a sample test template (on the left) and an example of a test generated from this template (on the right). The use of test templates thus separates the test-planning activity from the generator's development activity.

The template language consists of several types of statements: basic instruction statements, sequencing-control statements, standard programming constructs, and constraint statements. Users combine these statements to describe complex tests that capture the essence of the targeted scenarios, leaving out unnecessary details. Those details are left for the test generator to decide. This is done through pseudo-random decisions that can be biased by user directives in the template or through internal *testing knowledge* of the tool. Because the template does not fully specify the test, the same template can be used to create a large number of different test programs. On the very fast post-silicon platform, the number of test programs that can be run within a reasonable time is indeed very high. A good test template would therefore leave much room for variability.

The scenario targeted by the test template in Figure 2 was presented in [19] as a test for the *Coherency* rule of shared-memory models. This rule requires that no two processes can observe two accesses to the same memory location in a conflicting order. In the scenario, one process (process 0 in the figure) writes a sequence of monotonically rising values into some address, while the other processes read several times from the same address. If a value read is less than a value that was read earlier, this would indicate a violation of the Coherency rule.

The template begins by defining a template variable `addr` to keep the collision address and specifying a few general *biases* to affect the random selections made by the generator when selecting register and memory resources for the instructions. The template then defines the writing role for process 0 and the reading role for all the other processes (the `Process DEFAULT` statement), and specifies that these two roles should execute concurrently (the `Concurrent` statement).

The scenario for Process 0 includes 10 repetitions of (a) a store of some register to `addr`, (b) an increment of this register, and (c) a random instruction (either a random store, random page-crossing load, a load to `addr`, or some random arithmetic instruction). The generated test program on the right of Figure 2 shows the generated resource initializations, including the program-counters (`PC`) of the processes. The PC of process 0 is 500 and therefore its generated code starts at this address. The code for process 0 matches the template: process 0 continuously stores a register (`R5` in the test) into the same address (0x100 in the test), increments the register, and performs some random instruction.

Note that the specific memory location for the collision is not specified in the template. The template merely defines the `addr` template variable and uses it in the `Store` and `Load` instructions. Threadmill randomly selects the address, which could be different in different test programs generated from the template. In the example test program shown on the right of the figure, the selected address for the collision is 0x100. Similarly, the `reg` template variable is instantiated to register `R5` in the actual test. Threadmill makes such selections ran-

domly but biases the selections to create interesting verification events. The biasing can be controlled by user directives in the template, such as the request for register dependency at the top of the template, or the request for a page-cross access for the first `Load` of process 0 (which can indeed be seen to access address 0x22FFE in the test on the right).

Biasing can also be controlled automatically through testing knowledge embedded in the tool. Threadmill automatically tries to create collision events, such as a write-write false-sharing collision with Process 0's Store to 0x1EC40 and Process 1's store to 0x1EC48.

The scenario for the other processes has 10 repetitions of a sequence that includes (a) a load from `addr`, (b) a comparison of the loaded value to the previously loaded value, (c) a branch to an error procedure if a smaller value was observed (d) saving the loaded value for the comparison in the next loop iteration, and (e) a random store or load instruction. The generated test case includes corresponding code sections for the remaining processes of the system. The registers, memory addresses, and random instructions generated for the threads are different, but they all correspond to the general template for `Process DEFAULT` on the left of the figure.

The test-template language of Threadmill is very similar to the language of Genesys-Pro [13], IBM's primary test-generator for pre-silicon simulation-based core verification. To adhere to the requirements for simplicity and generation speed, several constructs that require long generation time, such as events, are not included in Threadmill's language.

A. Thread Irritation with Threadmill

Threadmill supports the generation of *thread irritation* scenarios as described in Section IV on silicon. There is no special feature that explicitely supports irritation, but the test-template is rich enough to enable the user to specify random irritation scenarios.

To do this, the scenario from one template serves as an "irritator" to a second "primary" scenario coming from another template. The two templates are combined in a single Threadmill exerciser image to create the irritation scenario. A single Threadmill exerciser can in fact generate scenarios that correspond to multiple template combinations. The user can define a multi-dimensional matrix of irritator and primary templates and let Threadmill generate test programs for every template combination in the matrix. The number of different template combinations that a single exerciser can handle in this manner can be very large, enabling the use of the exerciser for a much longer time in many different ways.

When combining templates to run concurrently, Threadmill ensures that the processes do not write over other processes' data and code areas. It does, however, enable threads from different scenarios to perform non-intrusive irritation such as to "falsely" share memory, collide using Load instructions, share address translation tables, and compete over computation resources such as a shared execution unit.

Combining multiple templates in a single exerciser image also has other benefits. First, loading an exerciser on the tested platform incurs an overhead - which can be decreased if multiple templates are handled by a single exerciser. In addition, a typical test performed by an exerciser on hardware can completed in relatively short time, in comparison to the test duration in simulation or acceleration environments. Thus, a basic exerciser for a single test template may quickly "run out of steam" with respect to its verification value. One way to get more value from available test templates is to use them in different template combinations.

Test Program Template	**Test Program**

Variable: addr
Bias: register-dependency
	Resource Initial Values:
	`0x100=7,`
	`0x22FFE=0x52766369,..`
	Process 0: PC=500, R5=0,..
	Process 1: PC=600, R1=17,..
	Process 2: PC=700, R1=42,..
	:

Concurrent	*Instructions (in memory):*

Process 0
Variable: reg
Repeat 10
Instruction: Store reg → addr
Instruction: Add reg ← reg + 1
Select
Instruction: Store ? → ?
Instruction: Load ? ← ?
Bias: page-cross
Instruction: Load ? ← addr
Group: Arithmetic
	`500: Store R5 → 100`
	`504: Add R5 ← R5+1`
	`508: Load R7 ← 22FFE`
	`50C: Store R5 → 100`
	`510: Add R5 ← R5+1`
	`514: Mul R2 ← R5, R7`
	`518: Store R5 → 100`
	`51C: Add R5 ← R5+1`
	`520: Store R2 → 1EC40`
	`524: Store R5 → 100`
	`528: Add R5 ← R5+1`
	:

Process DEFAULT
Variable: reg1, reg2
Instruction: Load reg1 ← addr
Repeat 10
Instruction: Load reg2 ← addr
Instruction: Compare reg2, reg1
Instruction: Branch-less-than ERR
Instruction: Move reg1 ← reg2
Select
Instruction: Store ? → ?
Instruction: Load ? ← ?
	`600: Load R7 ← 100`
	`604: Load R9 ← 100`
	`608: Compare R9, R7`
	`60C: Branch- less ERR`
	`610: Move R7 ← R9`
	`614: Store R9 → 1EC48`
	`618: Load R11 ← 100`
	`61C: Compare R9, R7`
	`620: Branch- less ERR`
	`624: Move R9 ← R7`
	`628: Store R7 → 23000`
	:

	`700: Load R5 ← 100`
	`704: Load R8 ← 100`
	`708: Compare R8, R5`
	`70C: Branch- less ERR`
	`710: Move R5 ← R8`
	`714: Load R14 ← 23000`
	`718: Load R8 ← 100`
	`71C: Compare R8, R5`
	`720: Branch- less ERR`
	`724: Move R5 ← R8`
	`728: Store R14 → 23004`
	:

	:

Fig. 2. Test template and corresponding test

B. Generating Random Collision Events

As described in Section IV, generating memory access collisions of various types is an important task of a test generator for multi-process systems. The collision generation must take into account the manner in which the collision will later be checked. As explained in Section III, Threadmill's main checking method is to run tests multiple times with the same resource initializations and verify that the final resource values are the same each time. This approach works fine for true-sharing read-read collisions and for all types of false-sharing collisions. However, the final value in the collision location in memory after a write-write collision depends on

the execution order of the write operations and may therefore be different in different executions of the same test. Similarly, a write-read collision may result in different values in the target register of the load, again depending on the order of the operations. For this reason, Threadmill does not check the memory used for write-write collisions, and the registers used in write-read collisions [3]. It still makes sense to generate these unchecked collision events because they may stress the hardware enough to cause failures that would be observed by other means (like machine checks).

Threadmill manages the collision generation and checking by a partition of the whole memory space into contiguous aligned 32-byte "sectors" (sectors are the basic units sent between the caches and the main memory). Every sector in memory is assigned an "owner" process, which is the only process allowed to write on it. Some sectors are designated as "unowned" and they can be written and read by all processes but are left unchecked. The owner process is also responsible to initialize its sectors before the test execution and to check their outcome at the end (only the sectors actually used in the test need to be initialized and checked). Some sectors are designated as read-only and some as read-write. On the IBM POWER7 processor (for which Threadmill was used), there are four sectors in a cache line. Thus, this sector ownership and access rights scheme enables all types of true and false sharing collision events.

The sector assignment needs to cover the whole memory space, so Threadmill only performs the assignment for a granule of 512 sectors (termed a "granule pattern") and then repeats the granule assignment pattern throughout memory. This scheme is shown in Figure 3. The whole memory space (as depicted at the bottom of the image) is split into contiguous aligned and identical ownership patterns of 512 sectors. For each of the sectors the granule pattern specifies the owner and access rights.

This mechanism sets access properties for the entire memory, but only a part of the memory is actually assigned for the random accesses made in a test. Specific areas in memory are assigned for the test code, and for the code and data areas of the exerciser itself. An exerciser image that combines several templates also allocates separate (non-contiguous) memory zones for the different templates. These constraints are handled by superimposing the corresponding restrictions on the access rights specified in the random granule pattern.

The task of generating the actual collisions is made much simpler after the ownership pattern is setup (as mentioned above, simplicity is an important requirement for an exerciser). Threadmill can now either select a random address and determine the collision properties from the pattern, or decide on a desired collision type (e.g., based on a user directive in the template) and use the pattern to come up with a random location. As explained in Section VI-A, the random decisions involved in generating a collision are made jointly

[3]Threadmill actually masks these registers with predictable values immediately following the collision

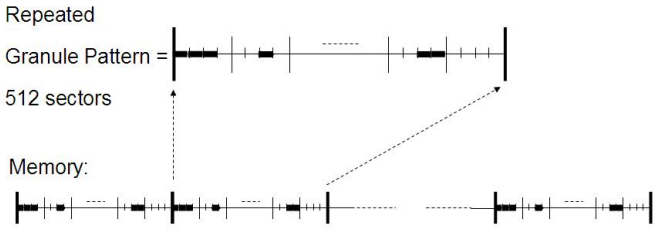

Fig. 3. Block Ownership and Access-Right Joint-Random Setting

by all the threads. The collaboration is managed by sharing the ownership pattern between the threads.

VI. Concurrent Generation Scheme

Threadmill's test generation component is designed to generate a random concurrent test program corresponding to the given test template, such as the concurrent program shown in Figure 2 based on the template in the same figure. The most straightforward way to do this is for a single process to generate the complete concurrent test program. This in fact is the common approach taken by pre-silicon test generators like Genesys-Pro [13], where each invocation of the (uni-process) test generator tool produces a new test program. The test programs are then collected and sent to simulation machines for execution and checking.

Threadmill, however, generates its test programs on the same parallel platform where the test program is executed. It can therefore take advantage of the parallelism offered by the execution platform to speed up the generation. In contrast, the uni-process generation approach is an inefficient use of computation resources; while the generating thread is creating the test program, the other threads remain idle. One way to utilize all the threads is to use them as separate test generators, where each thread independently generates a different concurrent test program corresponding to the template. Later, the same threads could also execute the collection of generated test programs. This solution increases the generation throughput but it does not reduce the latency until the first test can be executed.

Reduced latency is important when running on an acceleration platform. Acceleration is a technology for performing the design simulation using dedicated hardware. This speeds the simulation significantly when compared to performing the design simulation in software. The speed makes acceleration suitable for running post-silicon exercisers like Threadmill. However, acceleration is still much slower than the actual silicon platform. The number of cycles typically practicable for a single run on an accelerator could be too small to allow for a complete test program generation by a single process. It could however be enough to enable the parallel generation and execution of a test program.

Threadmill follows an alternative scheme in which all the threads collaborate to jointly generate every test program. The division of responsibilities is such that each thread generates its own role in the concurrent test program. When all the process parts of the test program have been generated, the

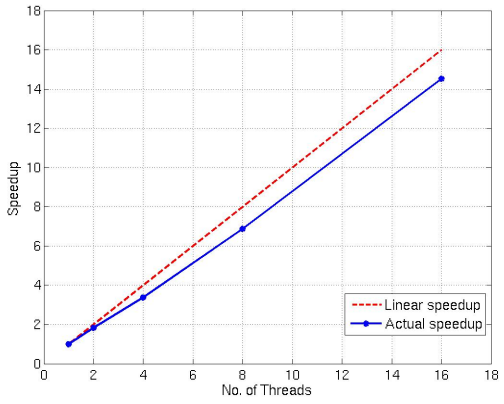

Fig. 4. Speedup with Concurrent Generation

threads synchronize and execute the generated program (every thread executing its own self-generated part). The benefit of this scheme is that it reduces the latency needed to get the first test (because the process of generating a complete test program is parallelized) while more or less preserving the throughput. In this sense, the solution is more scalable because the latency remains the same in larger systems.

Figure 4 shows the speedup gained with Threadmill's concurrent generation scheme relative to the scheme where one thread generates tests for all threads. The measurements were conducted by running a Threadmill image with corresponding number of threads on a POWER7 processor and counting the number of tests that were generated in 10 minutes. As can be seen in the figure, the concurrent generation scheme gives a speedup that grows almost linearly with the number of threads. This result together with a lower latency for the first test make the concurrent generation scheme a good approach for concurrent test program generation on silicon.

A. Managing Randomness

The Threadmill approach, in which every process generates its own part of the test, implies that some collaboration is needed between the generating processes when joint decisions need to be made about the generated test program - for example, when generating memory access collisions.

Threadmill's test template does not completely specify the test. It leaves extensive room for the test generator to make choices about various options, providing many opportunities to use the internal testing knowledge of the tool. These choices are made using pseudo-random decisions that can be biased with user directions in the test template.

Some of the random choices only affect the generating process' part of the test and can thus be made privately by each process. However, some of the random choices affect test portions of several threads. In these cases, the random selection must be made jointly by the affected processes so that all processes reach the same random choice.

Consider the test template example of Figure 2 in Section IV. The template defines a scenario where a writer thread repeatedly increments a counter repeatedly read by other threads. The template leaves the memory address of the counter undefined and allows Threadmill to randomly select the address (by using a template free variable - *addr*). The template also leaves the choice of the source register of Process 0's first store unspecified (by using a template free variable - *reg*). There is, however, a crucial difference between these two random choices: The choice of a source register for Process 0's store instruction can be made privately by Process 0 when it generates its own part of the test program, because this choice does not affect the test sections of the other threads. However, the random choice of a memory address for the shared counter must be made jointly by all the generating threads since it affects all their test sections.

Threadmdill thus needs to meet a basic requirement that is rare among concurrent applications: it must be capable of carrying out joint random decisions. A possible solution could be that whenever a joint decision is required, all the threads synchronize and arrive at the decision by means of communication (such as I/O or shared resources). One way to handle this is for one process to be distinguished as a "monarch". This process would then make all the joint random decisions and communicate the choices to all the other processes. The problem with these approaches, however, is that they require synchronization and communication for every joint decision. This typically incurs high performance costs, because threads that are ready to make and use the decision are delayed until other threads are also ready.

Joint random decisions are achieved using a pseudo-random number generator that is fed by a shared seed. This seed is separate from the seeds used by the private random decisions. In fact, the joint seed is used to generate the different private seeds for the processes so that all the random selections can be repeated when the exerciser run needs to be recreated during debugging. The original joint seed (which can be configured by the user) is shared by the processes by compiling it into the exerciser image during build time [4]. This is demonstrated in Figure 5, which shows the timelines of private and joint random numbers used by processes 0 and 1 for their respective random choices. It's crucial for all the threads to make the same decision; this is guaranteed by all of them using the same random number. The time in which each thread actually makes the decision is not important and therefore no synchronization is required. For this scheme to work, all the generating processes must make the same joint decisions in the same order. This is possible with Threadmill, since the joint decisions originate from the same test template, generated in the order in which the corresponding decisions appear in the template. The template language itself prohibits a branch on a private random choice that leads to a dependant joint choice.

A simple experiment we conducted showed that a Threadmill exerciser for 32 threads running on multi-processor hardware suffered a 40.41% decrease in performance when a single

[4]The image can be recompiled with a different seed to produce a different sequence of tests.

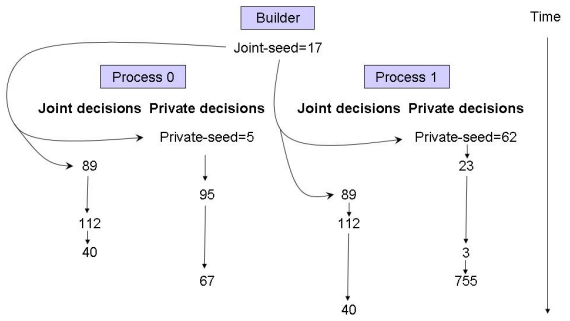

Fig. 5. Joint and Private Random Decisions

barrier was introduced to synchronize a random decision. Most of the overhead was spent in waiting for "the weakest link" to reach the barrier. By comparison, the computations required to compute a random seed are negligent.

Generating random collision events is one of the more important tasks carried out by a concurrent test program generator. An access collision event includes roles for several threads and thus requires joint random decisions to be made by the participating threads. These include the selection of the collision location and the type of collision required (for example true or false sharing, write-write, read-write, etc.). In Threadmill, the collision generation is based on the random setup of the ownership and access rights of the sectors in memory, following a random repeated pattern as described in Section V. All the generating threads determine the same pattern using the joint random seed. Thus, the ownership and access rights setting for the entire usable memory is shared among the threads. The threads can jointly select a random address or collision type and use the shared pattern to select the remaining properties of the collision event.

-

VII. Threadmill Experience on POWER7

The POWER7 processor implements the 64-bit IBM Power Architecture. Each POWER7 chip incorporates eight SMT processor cores with three levels of caches, memory and I/O controllers, and other support and management logic. The processor cores are out-of-order superscalar cores supporting up to four simultaneous threads. Each core contains twelve execution units shared between the threads. POWER7-based systems can contain up to 32 POWER7 chips, totaling 1024 threads running concurrently.

Threadmill was used in the post-silicon validation of POWER7 along with other tools, including similar exercisers generating concurrent programs. Threadmill focused on testing the multi-threaded aspects of the processor, using templates with intricate collision scenarios, for example, and deploying thread irritation. Threadmill's scalable generation scheme, in which every thread generates its own part of the concurrent test program, proved effective for the highly multi-threaded POWER7 systems.

Threadmill's primary inputs are test templates written in a language very similar to the test templates of Genesys-Pro, the main pre-silicon verification tool used for the POWER7 processor. This enabled a unified pre/post-silicon validation methodology where these two platforms complemented and improved each other [5]. For example, a bug found by Threadmill on a post- silicon platform could be more easily recreated with Genesys-Pro (with a test template similar to the one that exposed the bug) on a simulation platform where debugging is much easier.

Threadmill was also deployed on an accelerator platform before the actual silicon was available for bring-up. This platform enabled the early testing of Threadmill itself and the creation of a suite of test templates for bring-up with guaranteed coverage [5]. In addition, though Threadmill was mainly designed as a post-silicon tool, the accelerator platform enabled to use it as part of the pre-silicon verification effort. Once bring-up started, Threadmill was extensively used by the lab team running dozens of test templates on various POWER7 machine configurations. Overall, Threadmill ran for several hundreds of hours, and the early detection of bugs that it found was instrumental in enabling the clean shipment of POWER7-based systems.

An interesting bug detected by Threadmill during the early stages of POWER7 bring-up in the lab was related to thread starvation. The bug occurs when trying to get access to the store queue that holds pending store instructions waiting for completion in the memory hierarchy. In POWER7, the store queue is shared by all the threads in the same core. The bug was triggered in a thread-irritation scenario where three of the core threads were continuously dispatching long latency stores. The stores took a long time to complete, so new stores were always waiting to enter the queue. The fourth thread was also trying to perform a store and was starved from waiting to enter the queue due to thread asymmetry issues. Threadmill's thread irritation technique is particularly appropriate for catching these kinds of bugs because the irritator threads are good at stressing the execution units involved in the scenario (in addition to their ability to reach a large variety of timing situations). This bug was easily handled using a work-around based on existing logic on the processor by statically reserving at least one entry per thread in the store queue. The bug was later fixed by introducing an improved thread arbitration mechanism.

VIII. Conclusions

Modern high-end servers are highly parallel computers that concurrently run thousands of threads. Such servers are designed to support convenient programming models while providing high performance. This leads to extremely complicated implementations that are hard to verify. This verification challenge is magnified in post-silicon validation because of the high speed, long initialization time, and low observability that characterize post-silicon platforms.

In this paper, we presented some of the challenges of multiprocessing validation in post-silicon and the techniques and solutions used in Threadmill, an IBM post-silicon exerciser, to address these challenges. The challenges and solutions

presented in the paper focused on two areas: generating concurrent test programs for post-silicon validation, and the concurrent generation of such test programs. Threadmill was used to validate multiprocessing aspects of the POWER7 chip and POWER7-based systems. It found several multiprocessing-related bugs and contributed to the success of the POWER7 project.

REFERENCES

[1] B. Wile, J. C. Goss, and W. Roesner, *Comprehensive Functional Verification - The Complete Industry Cycle*. Elsevier, 2005.

[2] "International technology roadmap for semiconductors 2009 edition - design," Website, http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf.

[3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in test program generation for functional processor verification," *IEEE Design and Test of Computers*, vol. 21, no. 2, pp. 84–93, 2004.

[4] J. Storm. (2006) Random test generators for microprocessor design validation. Http://www.inf.ufrgs.br/emicro.

[5] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, and A. Ziv, "A unified methodology for pre-silicon verification and post-silicon validation," To appear in Proceedings of the 2011 Design, Automation and Test in Europe Conference, 2011.

[6] A. Adir, H. Attiya, and G. Shurek, "Information-flow models for shared memory with an application to the powerpc architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 5, pp. 502–515, 2003.

[7] A. Adir and G. Shurek, "Generating concurrent test-programs with collisions for multi-processor verification," in *Proceedings of the International High Level Design Validation and Test Workshop*, 2002, pp. 77–82.

[8] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[9] J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. Chu, M. L. Behm, J. R. Baumgartner, R. D. Peterson, and J. Abdulhafiz, "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 53–76, 2002.

[10] A. Adir, B. G. Hickerson, J. Ludden, and M. Rimon, "Advances in simultaneous multithreading testcase generation methods," in *Proceedings of the 6th Haifa Verification Conference*, ser. LNCS 6504. Springer-Verlag, 2010, pp. 146–160.

[11] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, "POWER7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, no. 2, pp. 7–15, 2010.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT-Press, 1999.

[13] M. L. Behm, J. M. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov, "Industrial experience with test generation languages for processor verification," in *DAC*, 2004, pp. 36–40.

[14] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI Magazine*, vol. 28, no. 3, pp. 13–30, 2007.

[15] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a constraint satisfaction formulation and solution techniques for random test program generation," *IBM Systems Journal*, vol. 41, no. 3, pp. 386–402, 2002.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy., "Memory consistency and event ordering in scalable shared memory multiprocessors." in *Annual International Symposium on Computer Architecrure*, 1990.

[17] S. Adve and M. D. Hill., "A unified formalization of four shared-memory models." *IEEE Transactions on Parallel and Disributed Systems*, vol. 4, no. 6, June 1993.

[18] C. May, E. Silha, R. Simpson, and H. Warren, Eds., *The PowerPC Architecture*. Morgan Kaufmann, 1994.

[19] W. W. Collier, *Reasoning About Parallel Architectures*. Prentice Hall, 1992.