Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors

Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir,
Gil Shurek, Vitali Sokhin, Avi Ziv
IBM Research - Haifa, Israel
{adir, maximg, shimonl, nahir, shurek, vitali, aziv}@il.ibm.com

ABSTRACT

Post-silicon validation poses unique challenges that bring-up tools must face, such as the lack of observability into the design, the typical instability of silicon bring-up platforms and the absence of supporting software (like an OS or debuggers). These challenges and the need to reach an optimal utilization of the expensive but very fast silicon platforms lead to unique design considerations - like the need to keep the tool simple and to perform most of its operation on platform without interaction with the environment.

In this paper we describe a variety of novel techniques optimized for the unique characteristics of the silicon platform. These techniques are implemented in Threadmill – a bare-metal exerciser targeting multi-threaded processors. Threadmill was used in the verification of the POWER7 processor with encouraging results.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Verification

Keywords

Post-Silicon Validation, Functional Verification, Stimuli Generation, Multi-Threading

1. INTRODUCTION

The Holy Grail of a single tape-out product seems to be drifting away for today's state-of-the-art processor and multiprocessor hardware systems. The functional verification of these systems involves hundreds of person years and requires the compute power of thousands of workstations [16]. Yet, the intricacy of modern microarchitectures and the complexity of the system topology makes it virtually impossible to eliminate all functional bugs in the design before tape-out. Statistics show that close to 50% of chips require additional unplanned tape-outs because of functional bugs [1]. Given this and facing condensing schedules, it is not unusual for project schedules to specify 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 to serve the development stage known as *post-silicon validation*. Traditionally, this stage is used to provide a stamp of approval for the design, aiming to expose any remaining electrical design flaws, catch the few functional bugs that escape into silicon, validate the system's performance, enable closure of firmware and system-software, and help characterize production issues. These days however, intermediate silicon prototypes need to play a more significant role in design lifecycle, and also serve as the next-level vehicles for functional verification of the system.

Post-silicon is getting a lot of attention in the hardware verification community in recent years [12, 13], but this attention is focused on the on-line checking and debugging capabilities of the silicon platforms (e.g., [2, 8, 11, 17, 9]). Very little has been published on post–silicon validation methodologies (e.g., [14, 3]) and other aspects of post-silicon validation, such as stimuli generation [3] and coverage [4].

Post–silicon validation is carried out by highly experienced teams of designers and domain experts, deploying a wide range of testing methods: from starter sets of simple tests, through massively pre-generated buckets of tests [14], specialized test-applications, exercisers [15], and layers of the system's software stack (i.e., OS, applications).

Post-silicon validation uses a variety of platforms, ranging from the first manufactured wafers to a series of experimentally constructed configurations of the system. These platforms are characterized by high manufacturing costs and relatively low availability. Also, when compared to simulation platforms, they offer real-time execution speed, but low dynamic *observability* into the system's state. A related limitation is the high overhead for loading and offloading memory and state information, in particular for a *bare-metal* system, where no OS services are available. Finally, some post-silicon platforms (e.g., on wafer) provide very limited memory space for programs and data.

The characteristics of the post-silicon platforms create challenges and impose tradeoffs that shape the way these platforms are used for functional verification. While post-silicon platforms offer a huge number of execution cycles, their low availability and high cost calls for high utilization in terms of maximizing the time spent in executing test-

DAC'11, June 5-10, 2011, San Diego, California, USA

cases and minimizing overhead. The overhead associated with loading a test image onto the platform may become a bottleneck, in particular for bare-metal platforms. Packaging a large number of pre-generated tests together would reduce the number of required loads and thus mitigate the overhead. However, this method requires sophisticated infrastructure to enable efficient synchronization and communication between a large test generation farm and the tested system. Alternatively, tests could be generated on the tested system by a post-silicon *exerciser [15]*, thus eliminate entirely the need to be loaded on the platform. An exerciser is an application that once loaded to the system, continuously generates test-cases, executes them, and checks their results. The on-platform test-generation process can significantly reduce the number of cycles left for the execution of test-cases, thus threatening to eliminate the advantage of the silicon prototype as a verification platform. Consequently, an on-platform test generation engine must be fast and light, and therefore simple, compared to technologies used for pre-silicon verification.

The requirement for simplicity goes beyond the utilization issue. The low observability makes hardware failures extremely hard to debug and therefore simple software must be used to ease the effort. In addition, we'd like to deploy the tool in the very early stages of the post-silicon validation effort when the OS cannot yet be run on the system and "complex" operations such as reading files from an I/O device are not supported. The limited observability, the need to maximize utilization, and the simplicity requirement suggest that on-platform, simple checking methods would be preferred. Reference models that are commonly used in pre-silicon verification are less appealing here: off-platform usage would require off-loading test execution results, while on-line usage would introduce complexity and significantly reduce the platform's utilization.

In [3] we presented a unified methodology for pre- and post-silicon verification and described Threadmill, an exerciser designed to support the post-silicon aspects of this methodology. Threadmill is a bare-metal exerciser, with a simple and fast pseudo random generation engine. Threadmill is directable, that is it enables fine user direction and control over the generated tests. It supports various checking methods, and provides mechanisms to assist in analysis of failures.

The description of Threadmill in [3] focuses on its architecture, and the methodology it supports, as well as some of the design decisions made to support the unified methodology. We now describe in detail some of techniques and mechanisms used to overcome the technical challenges induced by the platform and by Threadmill's design decisions.

Threadmill's design point induces several challenges. For efficiency and simplicity reasons, Threadmill avoids using techniques that are widely deployed by pre-silicon tools and are considered essential to enable fine user direction. It does not use constraint satisfaction techniques and is not supported by a reference model, neither to assist in test generation nor to support on-platform checking. The task of planning (or even predicting) the program's control flow becomes a challenge when a reference model is not available to support the generation process. For instance, we use a dual pass technique to generate a branch instruction. The generator plans the intended target address of the branch but the construction of the instruction itself is finalized during a preliminary test execution step, when the exact state of the required resources can be determined.

Some generation processes are too complex to run on the tested platform. One example is the generation of data for floating-point operands. Here, large buckets of operand-sets are generated off-platform, based on user directions, and are embedded in the exerciser's image. These sets of operands are later used by Threadmill while generating on-platform.

Threadmill is a distributed application, generating and checking distributively on each processor and thread. This scheme makes it simpler to achieve scalability on massively parallel systems, but makes the orchestration of correlated inter-thread, inter-processor, and system scenarios far more challenging. Implementing the generator engine as a distributed application means that multi-threaded interacting test-cases need to be created in a distributed way. Planning for interaction require synchronization or coordinated decisions between the generator's threads. An efficient way to achieve coordination over generation decisions, which should be random in nature, is to use a shared seed for the distributed pseudo-random decision procedures. Yet, certain user directions require that strong synchronization points, such as *barriers* and *critical sections*, are inserted to lock step the distributed generation processes. However, such procedures significantly reduce the efficiency of the scheme, and therefore user directions are analyzed to plan and minimize the use of generation-time synchronization points.

Threadmill uses *multi-pass consistency checking* [15], i.e., each test-case is executed multiple times ('passes') and the end-of-test values of some system resources (e.g., memory, registers) are compared for consistency. This method proved to cope well with the majority of bugs that escape into silicon. To support it, the generator must guarantee that generated scenarios lead to predictable results, at least for the checked resources. The effectiveness of this method greatly depends on existing and injected variability between passes (e.g., in thread scheduling or in operation modes of the system). Consistency checking is one way to cope with the challenge of checking in the absence of a reference model. The method is also readily amenable to Threadmill's distributed test-cycle control scheme. In addition to consistency checking, Threadmill supports user specified self-checking procedures. These are typically designed to verify predictable aspects of an abstract scenario outlined by the user.

We demonstrate the effectiveness of two of Threadmill's generation techniques (using off platform floating-point data generation and muti-threaded memory collision generation) through experiments.

The rest of the paper is organized as follows: Section 2 gives an overview of Threadmill. Sections 3, 4, and 5 describe how Threadmill overcomes specific challenges related to test generation on a post-silicon platform. Sections 6 and 7 describe Threadmill's checking and debugging techniques. Finally, we give our conclusions in Section 8.

2. THREADMILL OVERVIEW

Threadmill is a bare-metal, user-directable exerciser, with a simple and fast pseudo-random generation engine. The high-level tool architecture of Threadmill is depicted in Figure 1. Threadmill was developed to support a unified pre- and post-silicon verification methodology [3]. Its goal is to support a verification process guided by a verification plan by enabling the validation engineers to guide the exerciser

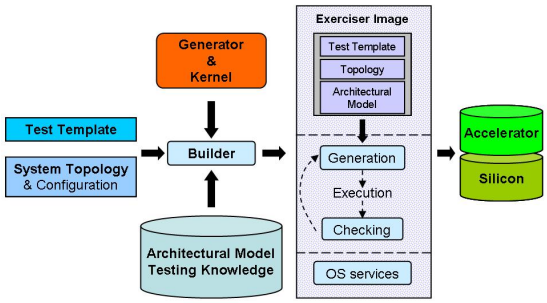

Figure 1: Threadmill architecture

through test-templates.

Threadmill was designed to address the unique characteristics of the post-silicon platform. To minimize interactions with the environment we chose to develop Threadmill as an on-platform exerciser. Thus once Threadmill is loaded on silicon it will generate, execute, and check tests "indefinitely". As described above, the lack of debugging capabilities coupled with the typical instability of the bring-up platform induces the requirement for a simple design. Therefore, Threadmill does not use sophisticated generation techniques like CSP [7], and does not rely on a reference model.

Threadmill was developed to be simple enough to also operate on hardware-acceleration platforms. Accelerators are hardware platforms dedicated to fast RTL simulation. The unique characteristics of the acceleration platform, as well as its implication on the verification methodology, are described in [3, 4] and are beyond the scope of this paper.

The main input to Threadmill is a test-template that specifies the desired scenarios from a verification plan. The test-template language of Threadmill is very similar to the language of Genesys-Pro [6], 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. The language enables verification engineers to write the test-templates as an activity separated from the generator's development activity. The language consists of four types of statements: basic instruction statements, sequencing-control statements, standard programming constructs, and constraint statements. Users combine these statements to compose complex test-templates that capture the essence of the targeted scenarios, leaving out unnecessary details. This allows them to direct the generator to a specific area. Other inputs to Threadmill are the architectural model, testing knowledge, and the system topology.

The Threadmill execution process starts with a builder application that runs off-line to create an executable *exerciser image*. 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. This scheme eliminates the need to access files or databases while the exerciser is running.

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 that is responsible for the exercising. The image created by the builder is then loaded onto the silicon/accelerator platform where the exerciser indefinitely repeats the process of 1) generating a random test-case based on the test-template, the configuration and the architectural model, 2) executing it, and 3) checking its results.

We designed Threadmill with the intention of focusing on the validation of multi-threaded designs (as implied by its name). It does this by generating multi-threaded test-cases. Since Threadmill runs on-platform, this implies that Threadmill itself is a distributed, multi-threaded, tool. Moreover, to maximize platform utilization and increase Threadmill's scalability, we implemented Threadmill's test generation component as a concurrent program, that is, every thread generates its own part of the test-case.

3. STATIC BRANCH GENERATION

We designed Threadmill's test generation component to be simple and fast. Therefore, we opted for a *static* test generator, i.e., one that does not make use of a reference model. Test generators that use a reference model (such as Genesys-Pro [6]) can leverage it to provide information about the expected state of the processor before and after the generation of each instruction. This information can be used to create more interesting events. For example, the generator can decide to select two registers that are expected to hold large numbers in order to trigger an overflow event when generating a *multiply* instruction. The problem then is how to generate valid and interesting test-cases without the ability to track the processor's state. A partial replacement to the reference model could be to explicitly reload resources, such as registers, with desired values right before their use within the test-case. However, this solution might interfere with the generation of the requested scenarios. For data-oriented events, such as divide-by-zero, a simple yet effective solution is to reserve registers to hold interesting values. Of course, the generator has to ensure that the reserved registers are not modified during the test-case.

Threadmill generates instructions in the order in which they will be executed. Therefore, it needs to know the outcome of a conditional branch to know at which address to generate the next instruction. This is hard to do without a reference model because it is impossible to evaluate the condition without knowing the expected values of the resources involved in the condition. One way to overcome this issue is to create the known machine state by inserting a well-known instruction sequence into the test so the outcome of the conditional branch is known at generation time. But this approach misses many dependency scenarios between the condition register setting and the branch.

To allow generation of an arbitrary instruction stream with conditional branches Threadmill completes the construction of the branches in two stages. First, as part of the general test generation flow, Threadmill decides on the desired condition result and target address of the branch. However, instead of placing the branch instruction into the test stream, it writes an *illegal* instruction form. This way, when the test-case is executed, the illegal instruction causes an illegal instruction interrupt. The handler of this interrupt calls a generation function dedicated to the completion of the branch instruction generation. This function is invoked during the actual run of the test-case, and can there-

fore observe the current state of the machine. This observed state is used to construct a condition expression that will evaluate to the desired result (as determined in the first stage). For example, suppose that at the first stage the generator decides that branch A should be taken. At run time, the illegal instruction interrupt is taken and the handler observes that register $R4$ contains a positive value. The handler now constructs a condition that would cause branch A to be taken if $R4$ is positive. The handler then replaces the illegal instruction with the generated branch and the execution of the test-case resumes. As described in Section 6, Threadmill executes each test-case multiple times for checking purposes. In the first execution (which serves as a kind of reference-model to the following executions) the branch only runs after the illegal instruction interrupt and the handler. In subsequent executions of the same test, the branch instruction is not delayed and runs right after its preceding instructions, without the handler. This approach, which defers some of the generation to the test-case execution, is very expensive performance-wise and is used sparingly, only in the cases where a reference model is crucial. The approach also requires execution of the same test-cases multiple times, but this multi-pass execution is a worthy cost because of its other benefits for checking purposes.

4. GENERATING FLOATING-POINT INSTRUCTIONS

Verification of the floating point unit (FPU) hardware implementation is known as an intricate problem. The numerous corner cases of the vast test space, coupled with the complexity of the implementation of floating point operations, turn the FPU verification effort into a unique challenge in the field of processor verification.

FPgen [5] is a test generation framework targeted toward the verification of the floating-point (fp) datapath, through the generation of data operands for fp instructions. The generated data can be directed to trigger a large variety of events related to the floating point computation, including intermediate and final data characteristics. It does this by leveraging various sophisticated stochastic and analytical techniques. Such techniques would be too complex for a computation by our on-platform test generator exerciser.

A simple approach could be to generate random floating point values for the instructions' inputs. However this technique has very limited coverage of the more intricate events related to fp instructions. Our approach uses FPgen "off-line" (i.e., not during the on-platform generation) to construct a large table of interesting data operands for the various fp instructions. Since this data generation is done off-line, we are less constrained by performance and complexity considerations. As part of the image building process, we select a random choice of data from the table of fp operands. This is then used to create a smaller table, which we incorporate into the exerciser image. The exerciser can now randomly select data for the generated fp instructions to trigger the interesting events targeted by FPgen.

In order to demonstrate the effectiveness of this method, we conducted the following experiment: we used Threadmill to generate tests with 10 different types of fp instructions (fadd, fsub, fmul, fdiv, fsqrt, in both single and double variations). We distinguished 22 types of fp number forms - 10 single forms and 12 double forms (such as normalized, denormalized, -infinity, etc.). We defined a coverage model that includes the occurrence of all the legal number forms in the input and outputs operands of the 10 fp instructions. This model includes 280 different legal events. We ran Threadmill in three modes. In the first mode we used random values for the inputs of the fp instructions; in the second mode we used only data collected from tables constructed with FPgen off-line as described above; in the third mode we used pre-generated data in 70% of the cases, while in the remainder random values are used. When only random inputs were used, 35.71% of legal events were covered. When only pre-generated data was used, 92.86% of legal events were covered. In the intermediate mode, where only 70% of instructions used pre-generated data, 97.5% of legal events were covered. While this may seem surprising, the reason for this is that FPgen focuses on generating *legal* interesting operand values, and therefore does not cover the illegal forms well (specifically, out-of-range numbers for single precision instructions). Therefore, combining pre-generated data with pure random values gives the best coverage results.

5. CONCURRENT GENERATION OF CONCURRENT TEST-CASES

The decision to make Threadmill's generation component a concurrent program (as described in Section 2) poses a significant challenge: Threadmill's generation component is required to create multi-threaded test-cases in a distributed way. Multi-threaded test-cases typically include some form of data sharing among the threads, for example, to better stimulate the hardware's memory coherence mechanisms. Therefore, the concurrently generating threads must come up with a shared set of addresses to be used in the respective thread parts of the test-case. Thus, some form of synchronization between the generating threads is needed. However, executing synchronizing sequences, such as a *barrier* or a *critical section*, can be very costly performance-wise.

To address this challenge we developed a mechanism that enables the threads to make joint random decisions with no synchronization. By "joint" we mean that all threads make the same decision, while keeping the decision random. This is achieved by a shared random seed. All the generating threads are provided with a common random number, created by the Threadmill builder. Every generating thread then makes all joint random generation decisions based on a random number generator initialized with this joint seed.

The joint random decision mechanism is heavily used in the generation of the addresses to be used by the threads when constructing their load/store instructions. In addition to jointly choosing the shared addresses, the generating threads must also jointly agree on the collision type (i.e., write-write, write-read, true- and false-sharing collisions, etc). Our approach is to decide on the type of collision allowed for every memory location. Thus the whole memory is mapped to the selected random collision types by using a joint random hash function, which is created using the joint random seed.

Without making these decisions jointly, the probability of generating colliding accesses among the threads is negligible (assuming the shared memory space is large). To demonstrate the effectiveness of the joint random collision generation scheme, we ran Threadmill on three configurations with $2GB$ of memory - two, four and eight threads. We

Threads	Read-Read	Write-Write	Read-Write	Total
2	67	64.4	129	260.4
4	200.4	228.6	387.6	816.6
8	476.8	527.6	833.2	1837.6

Table 1: Collision experiment results

used a test template specifying 50 load/store instructions per thread. We measured the number of generated collision events, where a collision event is defined as a pair of accesses by different threads that target (possibly different) locations in the same cache line (of size 128 bytes in our design). For example, suppose thread 0 loads twice from some cache line and thread 1 stores three times to the same cache line. This would be counted as 6 read-write collision events. The results of the experiment are presented in Table 1 where every row gives the average of 5 different runs.

6. CHECKING TECHNIQUES

The goal of checking is to detect failures as soon as possible and report as many details as possible to facilitate an effective debug process towards discovering the failure's root cause. In pre-silicon verification, the tests run on a testbench. The testbench incorporates checkers that detect most of the failures. These checkers, however, are not available when running on the post-silicon platform. Specifically, a reference model is not available to us due to its high complexity. We therefore have to devise and apply alternative checking methods when running Threadmill. While there are some checkers in the design itself that are monitored by the environment (e.g., machine went into an error state), our focus here is on checking done by the exerciser itself while it is running on the platform.

Threadmill employs a technique called *multi-pass consistency checking*. In this technique we execute each generated test-case several times (passes). After each pass we check that certain resources, including registers and memory areas, are consistent and have the same end-of-test values. In this way, we have the first pass essentially serve as a reference model. This consistency is guaranteed by restricting the test-case generation as follows. First, we do not randomly generate any instruction whose behavior is not fully predictable. This includes, for instance, the Load-and-reserve instruction, which does a load and additionally attempts to establish a reservation in memory. This attempt may succeed or fail depending on run-time conditions that involve other threads in the system, and cannot, in general, be predicted during test-case generation. Second, we forbid any write-write collisions on checked memory areas since we cannot predict the order between the colliding write accesses. Third, we must ensure that any data written to a checked resource is deterministic, in the sense that it has the same value in every pass. This is achieved either by having another checked resource serve as the source of the data or by simply using a constant. There is one exception to the latter restriction. A thread doing a load from a checked memory area that is written by another thread could be reading unpredictable data. In this case, Threadmill will inject into the test-case an additional instruction that overrides the target register with some constant, if that target register is checked. It is important to note that we do allow loads from unchecked memory areas as well, in which case the same overriding instruction as above is injected into the test-case.

It is beneficial to apply the multi-pass consistency checking technique with some variability between the different passes. Some variability is created automatically, for example due to page tables filling up and different cache states. Between passes Threadmill also modifies certain things that have no impact on the architectural behavior of the test-case, such as thread priority. It is important to note that the consistency checking technique targets bugs that can be detected due to different operation timing or order in the various passes. Specifically, the instructions of the different threads executing their parts of the test in parallel can be expected to be interleaved differently in different passes. Many multi-threading related bugs require a fine relative timing of the threads' operations. If the problematic timing occurs in one of the passes then the bug could be observed by the consistency check. In addition, the ordering of instructions executing in the pipeline of the same thread can also be different. However, some type of bugs cannot be detected. For example, a bug in the processor's datapath causing 1+1=3 will go undetected. However, we do not expect such bugs to reach the post-silicon stage at all, but rather be detected at a much earlier stage. Finally, this checking technique fits Threadmill very well. Threadmill generates multi-threaded test-cases by having each thread generate and execute its own part in the test-case. The consistency checking is done as follows. At the beginning of a pass, each thread initializes its registers, as well as the checked memory areas to which it is allowed to write. The thread then samples them for checking at the end of the pass. As a result, no extra synchronization barriers are needed, which is significant for achieving good platform utilization.

Self-checking tests is another checking technique used by Threadmill. In this case, the test scenario has some assertions embedded in it, and any violation of an assertion triggers a failure. The following example is based on a set of tests for shared memory multiprocessor by Collier [10]. The system is partitioned into n writer threads, numbered 0 through n-1, and remaining reader threads. Each writer thread i makes a series of atomic writes to a shared memory location, with an ascending series of values starting from i, i+n, i+2n, ..., and so forth. Each reader thread repeatedly makes an atomic read from the shared memory location, and figures out from which writer thread it originated from by using simple modular arithmetic. The reader thread then asserts that each series of read values originating from the same writer thread is monotonically non-decreasing.

7. DEBUGGING FAILING TESTS

Debugging failures on a post-silicon platform is a notoriously difficult problem. Many of the debugging aids that are available on simulation platforms cannot be supported on silicon because of the very limited observability into the state of the design. Furthermore, despite the fact that Threadmill itself is a piece of software, standard software debuggers cannot be used to debug it because they are typically too complex for the fragile state that the design is in during bring-up. The design's instability during bring-up can also cause unexpected behavior of Threadmill's core software components. It is thus often difficult to determine if a failure is due to a bug in the design or in Threadmill.

An important technique for debugging failures found by

Threadmill is to recreate the failing execution on a platform that *does* provide observability into the design - such as simulation or hardware acceleration. It is not enough just to re-execute the failing test-case since the bug could also have occurred during the test generation (which typically takes much longer than the test execution). In addition, a Threadmill image generates, executes, and checks a great number of test-cases, and a bug found through one test-case could be the result of a fault that occurred during the handling of a previous test-case. Restarting the image from the start on the simulation/acceleration platform is typically not possible because it could take days to simulate all the cycles executed on silicon before the bug occurs. Our approach is to restart the failing exerciser image on simulation/acceleration a few test-cases before the failure. We do this by reusing the corresponding random seeds that were used by the generating threads (as saved in the limited memory trace).

Another approach for test-case recreation is to take the test-template of the Threadmill image that caught the bug and use it with Genesys-Pro, the pre-silicon test generator counterpart of Threadmill. Genesys-Pro uses a test-template language similar to Threadmill's and can be used to generate test-cases for simulation with the hope that the bug re-occurs in one of them. This approach does not guarantee a precise recreation, but our experience shows that using the same test-template can get the design into the general area of the bug and improve the chances of re-triggering the fault under the same or similar circumstances.

8. CONCLUSION AND FUTURE WORK

The growing importance of post-silicon validation to the functional verification process increases the need to utilize silicon prototypes as a vehicle for functional verification. We presented Threadmill, a bare-metal exerciser targeting multi-threaded processors. Threadmill generates test-cases on platform, employing a variety of novel techniques optimized for the unique characteristics of the silicon platform.

Threadmill is an important enabler of the unified pre- and post-silicon verification methodology described in [3]. This methodology relies on a shared test plan and similar test template languages for the pre- and post-silicon test generators. Threadmill was used in the verification of the POWER7 processor chip. Results of this experience confirm our beliefs about the benefits of the various techniques described in this paper.

We are exploring further opportunities to leverage smart off-line pre-generation to improve the test quality – focusing on creating interesting address translation paths. This will enable Threadmill to become a more important player in areas such as caches and memory and I/O subsystems.

9. REFERENCES

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

[2] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for socs. In *Proceedings of the 43rd Design Automation Conference*, pages 7–12, July 2006.

[3] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, and A. Ziv. A unified methodology for pre-silicon verification and post-silicon validation. In *Proceedings of the 2011 Design, Automation and Test in Europe Conference*, pages 1590–1595, 2011.

[4] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *Proceedings of the 6rd Haifa Verification Conference*, pages 60–75, 2010.

[5] M. Aharoni, S. Asaf, L. Fournier, A. Koyfman, and R. Nagel. FPgen - a deep-knowledge test generator for floating point verification. In *Proceedings of the 8th High-Level Design Validation and Test Workshop*, pages 17–22, 2003.

[6] M. L. Behm, J. M. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov. Industrial experience with test generation languages for processor verification. In *Proceedings of the 41st Design Automation Conference*, pages 36–40, 2004.

[7] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Jouranl*, 41(3):386–402, 2002.

[8] K.-h. Chang, I. L. Markov, and V. Bertacco. Automating post-silicon debugging and repair. In *Proceedings of the 2007 international conference on Computer-aided design*, pages 91–98, November 2007.

[9] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pages 415–426, 2008.

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

[11] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. Backspace: formal analysis for post-silicon debug. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1–10, November 2008.

[12] S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Proceedings of the 47th Design Automation Conference*, pages 12–17, 2010.

[13] A. Nahir, A. Ziv, R. Galivanche, A. J. Hu, M. Abramovici, A. Camilleri, B. Bentley, H. Foster, V. Bertacco, and S. Kapoor. Bridging pre-silicon verification and post-silicon validation. In *Proceedings of the 47th Design Automation Conference*, pages 94–95, 2010.

[14] H. G. Rotithor. Postsilicon validation methodology for microprocessors. *IEEE Design & Test of Computers*, 17(4):77–88, 2000.

[15] J. Storm. Random test generators for microprocessor design validation, 2006. http://www.inf.ufrgs.br/emicro.

[16] D. W. Victor et al. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM Journal of Research and Development*, 49(4):541–554, 2005.

[17] I. Wagner and V. Bertacco. Reversi: Post-silicon validation system for modern microprocessors. In *Proceedings of the IEEE International Conference on Computer Design*, pages 307–314, 2008.