A Unified Methodology for Pre-Silicon Verification and Post-Silicon Validation

Allon Adir, Shady Copty, Shimon Landa,
Amir Nahir, Gil Shurek, Avi Ziv
IBM Research - Haifa, Israel
Email: {adir, shady, shimonl, nahir, shurek, aziv}@il.ibm.com

Charles Meissner, John Schumann
IBM Server and Technology Group
Austin, TX, USA
Email: {cmeissner, johnschu}@us.ibm.com

***Abstract*—The growing importance of post-silicon validation in ensuring functional correctness of high-end designs increases the need for synergy between the pre-silicon verification and post-silicon validation. We propose a unified functional verification methodology for the pre- and post-silicon domains. This methodology is based on a common verification plan and similar languages for test-templates and coverage models. Implementation of the methodology requires a user-directable stimuli generation tool for the post-silicon domain. We analyze the requirements for such a tool and the differences between it and its pre-silicon counterpart. Based on these requirements, we implemented a tool called Threadmill and used it in the verification of the IBM POWER7 processor chip with encouraging results.**

I. Introduction

The size and complexity of modern hardware systems have turned the functional verification of these systems into a mammoth task [1]. Verifying such systems involves tens or hundreds of person years and requires the compute power of thousands of workstations. But even with all this 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.

Post-silicon validation is not a new idea and has been used for many years in many places. It can be credited with the findings of many functional bugs that escaped pre-silicon verification. However, in general, functional verification methodology for pre-silicon is still more varied and mature than for post-silicon platforms. Very little is published on post-silicon verification methodologies (e.g., [2]), and most research in post-silicon validation has centered on on-line checking and debugging capabilities of the silicon platforms (e.g., [3]–[7]).

In recent years, we have seen more evidence that pre- and post-silicon verification cannot achieve their goals on their own; pre-silicon, in terms of finding all the bugs before tape-out, and post-silicon, in terms of finding the bugs that escaped pre-silicon. This creates an increasing need to bridge the gap between these two domains by sharing methodologies and technologies and building a bridge allowing easier integration between the domains.

We propose a unified methodology for pre- and post-silicon verification. The methodology extends the well-established *coverage-driven verification* (CDV) methodology [8] to the post-silicon verification domain. CDV is based on three main components: A verification plan comprising a large set of features in the *Design Under Verification* (DUV) that need to be verified; random stimuli generators directed towards the verification goals using test-templates [9] (i.e., general specifications of the desired test structure and properties); and coverage analysis tools [10] that detect the occurrence of events in the verification plan and provide feedback regarding the state and progress of the verification process.

The unified methodology calls for pre- and post-silicon to share the same verification plan, use similar languages to define test-templates for both domains, and use the same coverage models to measure the status and progress of the verification process. The methodology provides many advantages to users. It provides commonality and enables sharing of effort in the creation of the verification plan. It can also facilitate mutual feedback between the two domains, for example, in the activity of bug hunting.

While our goal is to create a unified methodology for the pre- and post-silicon domains, the major differences between the platforms dictate differences in the respective implementation of the methodology. One difference is that post-silicon platforms provide significantly higher execution speeds, which has implications on the verification tools that need to be adjusted for the best utilization of the available speed (see section IV). Another difference between the platforms is that common pre-silicon simulation platforms, including software simulators and hardware acceleration platforms (which perform the simulation on dedicated hardware and therefore offer much higher simulation speeds than software simulators) support a detailed level of *observability* into the state of the design as it is being tested [11], [12]. The silicon provides fewer opportunities to observe the behavior of the system. Therefore, bridging the gap between the pre- and post-silicon verification domains cannot be done blindly by copying methodologies and technologies from one domain to the other.

978-3-9810801-7-9/DATE11/©2011 EDAA

In fact, the limited observability of the silicon raises the need to use accelerators as a third platform in the proposed methodology, in addition to simulation and the silicon itself. Accelerators are fast enough to run simple post-silicon tools, and are observable enough to allow coverage measurement. This allows us to measure coverage of the post-silicon tools in a silicon-like environment.

The focus of the paper is the stimuli generation aspect of the unified methodology. We show how many of the concepts and technologies that made constrained random test generators successful in pre-silicon verification can be adapted to the post-silicon domain. Specifically, we discuss how a user-directable tool that uses declarative test-templates can be used in post-silicon validation. The differences between the pre- and post-silicon platforms mean that we cannot simply convert a successful pre-silicon generator into a post-silicon tool. For example, to avoid spending too much time on generation compared to execution of the tests, the heavy sophisticated generation engine used in pre-silicon generators needs to be replaced with a lighter and much faster engine. In addition, to avoid long loading time, the post-silicon solution needs to be in the form of an *exerciser*, that is, a tool that continuously generates test cases, executes them, and check their results.

We implemented the proposed methodology in a tool called Threadmill [13]. Threadmill is a bare-metal exerciser, i.e., an exerciser that operates on the silicon itself without the support of an operating system. Threadmill fulfills the above requirements; it is a directable random generator with a simple and fast generation engine. In addition, it meets other requirements of exercisers, such as providing mechanisms to assist in checking. Threadmill also addresses technical challenges, such as how to replace the reference model, which is an essential part of both generation and checking in pre-silicon.

The proposed unified methodology and Threadmill were used for the first time in the verification and bring-up of the IBM POWER7 processor with encouraging results. The use of Threadmill (and other exercisers) on accelerators provided significant help to the pre-silicon verification effort in finding many bugs, some of them severe. The coverage feedback at this stage provided a means for building a good set of test-templates for the bring-up team, which helped ensure a smooth bring-up and reduced the number of escapes from the first tape-out [11]. In addition, the user's ability to direct Threadmill through test-templates aided in recreating some bugs found on silicon in a pre-silicon environment, which helped analyze and fix these bugs.

The rest of the paper is organized as follows: in Section II, we describe the main properties of a pre-silicon stimuli generator, as a reference. In Section III, we define the requirements for a directable post-silicon stimuli generator. Section IV describes the proposed unified methodology. Section V describes Threadmill, our post-silicon directable exerciser. Section VI shows our experience with the unified methodology and Threadmill in the verification of the POWER7 processor. Section VII concludes the paper.

Test Program Template	**Test Program**
Variable: addr = 0x100	*Resource Initial Values:*
Variable: reg	`R6=8, R3=-25,.. R17=-16`
Bias: register-dependency	`100=7, 110=25,.. 1F0=16`
	Instructions:
Instruction: Store R5 → ?	`500: Store R5 → FF0`
Repeat (addr < 0x200)	`504: Load R4 ← 100`
Instruction: Load reg ← addr	`508: Sub R5 ← R6-R4`
Select	`50C: Load R4 ← 110`
Instruction: Add ? ← reg + ?	`510: Add R6 ← R4+R3`
Bias: sum-zero	:
Instruction: Sub ? ← ? - ?	`57C: Load R4 ← 1F0`
addr = addr + 0x10	`580: Add R9 ← R4+R17`

Fig. 1. test-template and corresponding test

II. Pre-Silicon Stimuli Generation

A pre-silicon stimuli generator has to provide the user with the ability to specify the desired scenarios in some convenient way, and produce many valid high-quality test cases according to the user's specification. These scenario specifications - known as *test-templates* - are written in a language that should enable an easy and accurate way for specifying the scenarios from the verification plan. Figure 1 shows an example of a test-template that defines a table-walk scenario (on the left) and an example of a test generated from this template (on the right). The test-template is written in the test-template language of GenesysPro- [14] - IBM's well-established test generation tool for the functional verification of processors using a software simulator platform. The rest of this section describes Genesys-Pro's approach to test generation.

The scenario of figure 1 starts with a `Store` and then a sequence of `Loads`, each followed by either an `Add` or a `Sub` instruction. The memory locations accessed by the `Load` instructions are contiguous in memory as seen in the Resource Initial Values section of the test (addresses 0x100–0x1F0). This is managed by a test-template variable `addr`.

The use of test-templates thus separates the test planning activity 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 describe complex tests that capture the essence of the targeted scenarios, leaving out unnecessary details.

The generated test cases must be valid according to the processor's architecture, and satisfy the user's request specified in the template. In addition, they should also be different from each other as much as possible. This is done by specifying the rules determining the validity of a test case, as well as the user's requests as constraints. The generator then produces a test case by random sampling of the solution space to the resulting constraint satisfaction problem [15].

The distribution of the generated tests should not be uniform, as we want to favor tests that include interesting verification events (e.g., register dependency, memory collisions), especially ones that are extremely unlikely to occur under uniform distribution. This is done by having knowledge embedded in the generator, allowing it to bias random

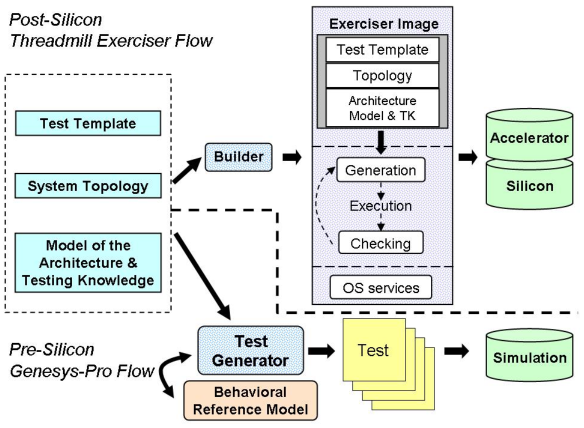

Fig. 2. Threadmill Vs. Genesys-Pro tool flows

decisions towards stimuli that cause interesting events [15]. This *testing knowledge* defines the interesting verifications events, including the stimuli that trigger them. As the stimuli for some interesting events depend on the processor's state, the generator also employs a reference model of the DUV, simulating on it every generated instruction. This way the generator maintains an accurate view of all the architectural resources, and takes them into account during the generation of interesting events. This scheme is shown in the lower part of Figure 2.

Genesys-Pro has been in use by IBM for over ten years. It has proven to be effective in meeting users' requirements, enabling them to write test-templates implementing the core verification plans of IBM's complex processors [16].

III. Post-silicon Stimuli Generation

The characteristics of the post-silicon platform and the differences between it and pre-silicon software simulators dictate differences in the best ways to use this platform for functional verification in general, and to stimuli generation specifically. The first important characteristic is the long loading and initialization time of the silicon. Consider, for example, a test-case of 10,000 instructions. Such a test-case takes less than one millisecond to complete on an advanced processor. On the other hand, preparing the processor for the execution (i.e., performing a power-on sequence, loading the test-case to memory, etc.) can take more than a minute. This results in a execution utilization of less than 0.002%.

As a result, post-silicon validation tends to rely more on longer running solutions. One commonly used solution of this type is the use of available applications, including operating systems. Another solution is the use of exercisers, which are programs that run on the DUV and "exercise" it by testing interesting scenarios. An exerciser is a self-contained solution. It generates the test-cases, runs them, does the checking, and contains OS services required by the test-cases. The exerciser runs in an endless loop which makes it a good post-silicon solution since it is only loaded once on the DUV.

The difference in speed between pre- and post-silicon platforms affects the way the platform cycles are utilized. While pre-silicon test-generators strive to use the scarce simulation cycles as effectively as possible by using extensive testing knowledge to generate very high-quality test-cases, the silicon platform is anything but short on run-time cycles. Therefore to divert the resources from generation to execution, a post-silicon exerciser should spend less effort in generating precise interesting scenarios and instead generate more general test-cases using a lightweight generator. This loss of precision is compensated by an increase in number of tests generated.

Another area where the pre-silicon platforms differ from the silicon platform is the level of observability. This has a deep impact both on the ability to perform checking and the ability to measure coverage. Pre-silicon checking techniques include for example scoreboards, and assertions [1] which depend on a high level of observability into the design which is not available in typical silicon platforms. Similarly, the ability to measure coverage on the silicon is very limited due to the low observability. We overcome this problem by running our post-silicon exerciser on the acceleration platform where coverage can be measured as detailed in Section IV.

Another requirement for a post-silicon exerciser is simplicity. Hardware failures are hard to debug and therefore simple software must be used to ease the effort. In addition, we'd like to deploy the tool during very early stages of the post-silicon validation effort when OS can't be run on the DUV and operations such as reading files from an I/O device are not supported. Another requirement is to keep the ratio between time spent in generating and checking the test-case and the time spent running it as low as possible. For example, while embedding a reference model in an exerciser could improve its checking capability, the reference model itself is complicated software that would significantly reduce the platform's utilization. Examples of checking techniques that don't use a reference model are mentioned in Section V.

IV. A Unified Verification Methodology

To better integrate post-silicon validation to the overall verification process and improve its synergy with pre-silicon verification, we need a unified verification methodology that is fed from the same verification plan source. A key ingredient for the success of such methodology is providing common languages for the pre- and post-silicon aspects of it in terms of test specification, progress measure, etc. Figure 3 depicts such a methodology. This verification methodology leverages three different platforms: simulation, acceleration, and silicon. The methodology requires three major components: a verification plan, directable stimuli generators suited to each platform, and functional coverage models. Note that important aspects in any verification methodology, such as checking, are omitted from the figure to maintain focus on the main aspect of our contribution, namely stimuli generation.

The verification plan includes a long list of line-items, each targeting a feature in the DUV that needs to be verified. Each such feature is associated with coverage events that the

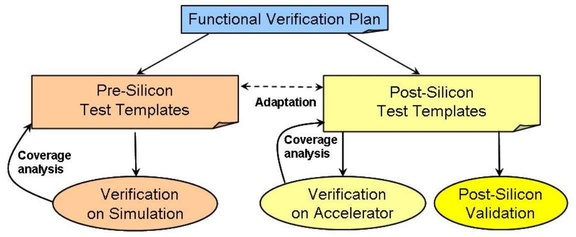
Functional Verification Plan
Pre-Silicon Test Templates
Adaptation
Post-Silicon Test Templates
Coverage analysis
Coverage analysis
Verification on Simulation
Verification on Accelerator
Post-Silicon Validation

Fig. 3. A unified verification methodology

verification team expects to observe during the verification process and the methods to be used to verify the feature. The verification plan is implemented using random stimuli generators that produce a large number of test-cases, and coverage tools that look for the occurrence of events in the verification plan. The random stimuli generators are directed towards the verification goals by using *test-templates*. The test-templates allow the generators to focus on areas in the DUV ranging from large generic areas, like the floating-point unit, to very specific areas, like a bypass between stages of the pipeline. Coverage analysis identifies gaps in the implementation of the plan. Its feedback is used to modify test-templates that do not fulfill their goals, and create new ones.

Extending this methodology to post-silicon validation is difficult because the limited observability of the silicon does not allow to measure coverage on the silicon. To overcome this problem, we leverage the acceleration platform to measure coverage of post-silicon tools. To take advantage of the coverage information collected by the accelerators and use it in the post-silicon, shortly before first silicon samples come back from the fab, a regression suite of exerciser test-templates is created based on the coverage achieved on the accelerators. This regression suite is then used to continue the verification process on the silicon platform.

With the unified methodology, we attach to each of the line-items in the verification plan one or more target platforms on which it will be verified. These line-items are converted to test-templates in the languages of the generation tools used by each platform. A key ingredient for the success of the unified methodology is similar operation of the stimuli generators. In this sense, we would like the generators to use the same test-template language, and when provided with the same test-template, we would like the tools to produce similar (though not identical) test-cases. Of course, the different platforms provide different opportunities and put different constraints and requirements on the generation tools, but whenever possible, there are advantages to having similar tools. First, the pre- and post-silicon teams can share the task of understanding the line-items in the verification plan and planning ways to test them. In addition, the common language allows for easier adaptation of test-templates from one platform to another. For example, when a bug is detected on the silicon platform, narrowing down the test-template and hitting it on the simulation platform eases the root-cause analysis effort.

It's important to note that the differences between platforms also dictate differences in the way test-templates are written for pre- and post-silicon tools. A test-template could be very specific and describe a small set of targeted tests or it could be more general leaving more room for randomization. The validation engineer writing test-templates for a post-silicon exerciser must bear in mind the fact that the test-template is used to generate a huge number of test-cases and get many processor cycles. To effectively use these test cycles, the test-template must allow for enough interesting variation. A test-template that is too specific will quickly "run out of steam" on silicon and start repeating similar tests. A pre-silicon test-template on the other hand would typically be more directed to ensure that the targeted scnearios are reached within the fewer cycles available on simulation.

V. Threadmill

Threadmill was developed to enable the unified methodology described in Section IV, i.e., to support a verification process guided by a verification plan by enabling validation engineers to guide the exerciser through test-templates. The high-level tool architecture of Threadmill is depicted in Figure 2, along with the flow of Genesys-Pro [9] - the pre-silicon test generator tool described in Section II.

Like Genesys-Pro, the main input to Threadmill is a test-template that specifies the desired scenarios. For reasons mentioned in Section IV, the templates used for pre- and post-silicon tests have different characteristics. The test-template language of Threadmill is very similar to the language of Genesys-Pro. However, to adhere to the simplicity and generation speed requirements, several constructs that require long generation time, such as events, are not included in Threadmill's language. Other inputs to Threadmill include the architectural model and testing knowledge and the system topology. Again, for simplicity reasons, many testing knowledge items that are included in Genesys-Pro models are not used by Threadmill.

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 that 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 executable image created by the builder is then loaded onto the silicon platform where the exerciser indefinitely repeats the process of generating a random test case based on the test-template, the configuration, and the architectural model, executing it, and checking its results.

In the case of Genesys-Pro, the test generation process is carried outside of the simulation environment (say on a dedicated server) and only the generated tests are loaded and run on the simulation platform. Simulation cycles would be too slow to allow generation during simulation. The "off-line" generation, on the other hand, can afford to spend time on sophisticated generation and checking, for example, by using a reference model as seen in Figure 2 for Genesys-Pro. Threadmill's test generation component was designed to be simple and fast. The generation is therefore *static*, i.e., without the use of a reference model. Reference models provide the generator information about the state of the processor before and after the generation of each instruction. This information is used for checking but also to create more interesting events. Reloading resources, such as registers, can be a partial replacement to the reference model, but this solution potentially interferes 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.

Execution of the same test case multiple times is used as a partial replacement for checking done by the reference model. This is done by comparing certain resource values, such as registers and part of the memory, for consistency in different executions of the test case. Running the same test case multiple times may result in different results even when bugs are not present. For example, when several threads write to the same memory location, the final value at this location depends on the order of the write operations. This requires that certain mechanisms be implemented in the generator to restrict the number of unpredictable resources. Although the multi-pass comparison checking technique is limited, it has proven to be effective when control-path-oriented bugs, or bugs that reside in the intersection of the control- and data-paths, are concerned. 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 maintains the predictability of the compared resources. This can be done, for example, by changing the machine mode, or changing thread priorities.

VI. POWER7 Experience

The proposed unified methodology and Threadmill were used in the verification of IBM's POWER7 processor. 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 that are shared between the threads. This section highlights some of the results and lessons learned from this first use of the unified methodology and Threadmill in a large industrial project. It focuses on the core verification of the POWER7.

POWER7 was verified using three platforms: simulation, acceleration, and silicon. Traditional pre-silicon simulation-based verification was used throughout the lifetime of the project with the goal of revealing as many defects as possible. As soon as the core was sufficiently stable, the *Exercisers on Accelerators* (EoA) phase started and was executed in parallel to simulation. Following every tape out, a post-silicon validation phase started. This process was accompanied by a supporting effort on both the simulation and acceleration platforms to assist the analysis of bugs that were initially detected on the silicon and the verification of the bug fixes.

The influence of the unified methodology on the POWER7 verification started in the EoA phase. Exercisers and accelerators have been used in previous projects [16], but POWER7 was the first time in which the EoA effort was tightly integrated in the overall verification effort of the core. Each exerciser was assigned several line items in the core verification plan and was asked to provide test cases that cover the events associated with these line items. The coverage measured on the accelerators was incorporated in the project coverage reports together with unit and core simulation coverage. Accelerator coverage was also used to harvest high-quality test-templates for post-silicon validation.

Threadmill was not the only exerciser used in POWER7. Several other exercisers were also used, and each of the exercisers was assigned its own items in the verification plan, based on its capabilities. Each exerciser created test cases for the verification items in its own way. Some exercisers used modification of their base software; others used parameters, tables, and macros for that purpose. Threadmill was the only exerciser that used declarative test-templates. This provided Threadmill with several advantages. First, it provided a convenient way for tool users to specify the desired scenarios. It also allowed easy sharing of knowledge between Threadmill and Genesys-Pro due to the use of a similar language. Finally, it provided a means for easier transfer of bug information between the platforms for bug recreation and fix validation.

The use of EoA provided several advantages to POWER7. First, the EoA process caught some high impact bugs. One such bug detected by Threadmill shortly before the first tape out was a show-stopper bug that would have totally undermined the first post-silicon validation process and would have required an additional tape out. In terms of coverage, the EoA effort obtained results that are comparable with the core simulation effort. Table I shows the coverage results shortly before the first tape out for unit simulation, core simulation, and EoA. The table shows that for control-oriented units, such as the fetch unit (IFU) and sequencing unit (ISU), EoA coverage is almost similar to the core simulation coverage. This is an indication of the exercisers' ability to reach interesting scenarios even in a small number of acceleration cycles (compared to the silicon platform).

The shared verification plan and the EoA effort gave a boost to the post-silicon validation phase. The organized verification plan and the harvested test-templates provided the bring-up team with a good starting point to their effort and a means to track progress. The heavy use of the exercisers in EoA also helped improve their readiness for the bring-up. All this supported a much shorter bring-up phase. In addition,

DUV Unit	Unit Sim	Core Sim	EoA	Total
IFU	96.79	96.77	94.99	98.65
ISU	96.48	92.49	92.78	97.42
FXU	99.60	84.72	85.85	99.85
FPU	97.44	98.15	90.20	99.58
LSU	94.33	91.04	85.32	98.66
PC	92.51	76.95	55.23	93.51
Core Total	96.18	92.78	88.70	98.06

TABLE I
POWER7 COVERAGE RESULTS

compared to previous projects, a larger percentage of complex bugs were found earlier in the bring-up process and by bare-metal tools, which are easier to debug.

The unified methodology and the similar test definition languages of Genesys-Pro and Threadmill provide another advantage in the ability to hasten root-cause analysis of bugs and validation of bug fixes. This process produced a synergy between platforms where each task in the recreation, analysis, and fix validation of the bug was performed on the most suitable platform. For example, the relative speed of accelerators compared to simulators was used to look around a complex bug found by Genesys-Pro in simulation. This was done by generalizing and adapting the test-template that produced the bug to Threadmill. Another example started with a bug that was found on silicon. The test case that found the bug (that was not found by Threadmill) and basic observations from the silicon caused the designer to speculate on a reason for the bug. The next step was to create test-templates for Threadmill (on the accelerator) to test that theory. Once the bug was recreated on the accelerator, the test-template was narrowed down and the bug was recreated in simulation. Finally, after the bug was fixed, all the test-templates created for its recreation were used on the appropriate platforms to ensure that the fix was correct.

The success of the unified methodology was evident in the bug shift from silicon to acceleration and in the productivity improvements that it gave to the post-silicon, acceleration, and pre-silicon teams. The verification of POWER7 is considered by experts to be the best executed in IBM in recent years. The low number of bugs that escaped to silicon and escaped from the first to the second tape out provides good supporting evidence of this. We believe that the wour work has made a significant contribution to POWER7's success.

VII. Conclusions

The growing importance of post-silicon validation to the verification process increases the need for synergy between the pre- and post-silicon verification methodologies and technologies. This paper presented a unified verification methodology based on a common verification plan and coverage models and that uses similar test-template languages. To implement the methodology, we transferred and adapted the pre-silicon stimuli generation methodology into post-silicon. In this transition, we maintained the concepts of directed random stimuli generator that is controlled via declarative test-templates. Other aspects, such as the trade-off between generation smartness and generation speed, were adapted to better fit the requirements of the post-silicon domain.

The resulting tool, a bare-metal exerciser called Threadmill, was used in the pre- and post-silicon verification of the POWER7 processor chip. Results of this experience confirm our beliefs about the benefits of the increased synergy between the pre- and post-silicon domains and of using a directable generator in post-silicon validation.

We see several ways to enhance the work described in this paper. First, we would like to extend this "bridging the gap" concept to other aspects of functional verification, such as checking and coverage. In addition, we are working to incorporate more testing knowledge into Threadmill. For example, we are working to improve Threadmill's ability to create interesting address translation paths. This will enable Threadmill to better exercise areas such as cache coherency and the memory and I/O subsystems.

References

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

[2] H. G. Rotithor, "Postsilicon validation methodology for microprocessors," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 77–88, 2000.

[3] 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*, July 2006, pp. 7–12.

[4] 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*, November 2007, pp. 91–98.

[5] 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*, November 2008, pp. 1–10.

[6] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *ICCD*, 2008, pp. 307–314.

[7] K. Chen, S. Malik, and P. Patra, "Runtime validation of memory ordering using constraint graph checking," in *HPCA*, 2008, pp. 415–426.

[8] H. B. Carter and S. G. Hemmady, *Metric Driven Design Verification: An Engineer's and Executive's Guide to First Pass Success*. Springer, 2007.

[9] 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.

[10] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. Springer, 2004.

[11] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann, "Reaching coverage closure in post-silicon validation," in *Proceedings of the 6th Haifa Verification Conference*, 2010.

[12] J. Darringer *et al.*, "EDA in IBM: past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1476–1497, December 2000.

[13] Building a bridge: from pre-silicon verification to post-silicon validation. [Online]. Available: http://es.fbk.eu/events/fmcad08/presentations/tutorial_moshe_levinger.pdf

[14] 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.

[15] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," in *AAAI*, 2006.

[16] D. W. Victor *et al.*, "Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems," *IBM Journal of Research and Development*, vol. 49, no. 4, pp. 541–554, 2005.