Improving Post-silicon Validation Efficiency by Using Pre-generated Data

Conference Paper · November 2013

DOI: 10.1007/978-3-319-03077-7_12

6 authors, including:

Anatoly Koyfman
IBM
19 PUBLICATIONS **125** CITATIONS

SEE PROFILE

Improving Post-Silicon Validation Efficiency by Using Pre-Generated Data

Wisam Kadry, Anatoly Koyfman, Dmitry Krestyashyn,
Shimon Landa, Amir Nahir, and Vitali Sokhin

IBM Research – Haifa, Israel,
{wisamk, anatoly, krest, shimonl, nahir, vitali}@il.ibm.com

Abstract. Post-silicon functional validation poses unique challenges that must be overcome by bring-up tools. One such major challenge is the requirement to reduce overhead associated with the testing procedures, thereby ensuring that the expensive silicon platform is utilized to its utmost extent. Another crucial requirement is to conduct high-quality validation that guarantees the design is bug-free prior to its shipment to customers. Our work addresses these issues in the realm of software-based self-tests.
We propose a novel solution that satisfies these two requirements. The solution calls for shifting the preparation of complex data from the runtime to the offline phase that takes place before the software test is compiled and loaded onto the silicon platform. This data is then variably used by the test-case to ensure high testing quality while retaining silicon utilization.
We demonstrate the applicability of our method in the context of bare-metal functional exercisers. An exerciser is a unique type of self-test that, once loaded onto the system, continuously generates test-cases, executes them, and checks their results. To ensure high silicon utilization, the exerciser's software must be kept lightweight and simple. This requirement contradicts the demand for high-quality validation, as the latter calls for complex test generation, which in turn implies extensive, complex computation. Our solution bridges these contradicting requirements.
We implemented our proposed solution scheme in IBM's Threadmill post-silicon exerciser, and we demonstrate the value of this scheme in two highly important domains: address translation path generation and memory access management.

1 Introduction

In today's state-of-the-art multi-processor and multi-threaded hardware systems, eliminating all functional bugs in the design before the first tapeout is virtually impossible. This is due to the intricacy of modern micro-architectures and the complexity of the system topology. Consequently, the development stage known as *post-silicon validation* [17, 19, 25], which is responsible for catching any remaining functional bugs before they escape to the field, is of growing importance.

Post-silicon validation uses implementations of the system on silicon running at realtime speed. These platforms are relatively scarce and have high manufacturing costs. Moreover, when compared to simulation platforms, they offer realtime 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. These characteristics create challenges and impose tradeoffs that shape the way post-silicon platforms are used for functional validation. While post-silicon platforms offer a huge number of execution cycles, their scarcity and high cost call for high machine utilization in terms of maximizing the time spent in executing test-cases and minimizing overhead. The overhead associated with loading a test-case image onto the platform and offloading its results may become a bottleneck. Under such circumstances, the silicon platform would be idle for a portion of the time.

A prominent technique of driving stimuli (as well as performing checking) in post-silicon validation is that of Software-Based Self-Tests [26], or SBSTs. SBSTs are actually simple programs that run on the silicon under test to verify its behavior. This approach is highly advocated as it requires no additional testing-specific instrumentation of the hardware.

A unique type of SBSTs is that of bare-metal exercisers [5, 29]. An exerciser is an SBST that, once loaded onto the system, continuously generates test-cases, executes them, and checks their results. (We omit the details involving the compilation of the exerciser into a loadable executable image.) The generated test-cases must be valid programs for the hardware threads to execute. They must also be sufficiently complex, such that they stress the design and trigger meaningful events in various areas.

We are, therefore, faced with two seemingly conflicting requirements when designing the stimuli generator of a post-silicon validation exerciser. On the one hand, achieving high machine utilization requires the exerciser's stimuli generation engine to be fast and light. On the other hand, achieving high validation quality requires that very same engine to be capable of generating complex test-cases. Moreover, if a single exerciser image is to be loaded and run for a long time on silicon at realtime speed, sufficient variability must exist among the generated test-cases. Consequently, these requirements constitute a considerable challenge to the design of a post-silicon exerciser.

Finally, an exerciser has to be simple. The low observability makes failures extremely difficult to debug, and therefore, simple software must be used to ease the effort. We also want to deploy the exerciser in the early stages of the post-silicon validation efforts 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.

Contributions. In this work, we address the problem of conducting high-quality design validation without harming the utilization of the expensive silicon platform under test in the realm of SBSTs. The novelty of our solution lies in preparing the input data off the platform and efficiently integrating it into the SBST. As the input data is prepared off platform, we can leverage sophisticated (and highly complex) techniques to ensure the data is of high quality. The data is formatted and structured in a way that enables the SBST to easily access it with minimal overhead.

We demonstrate the effectiveness of the proposed method in the context of bare-metal exercisers. Specifically, we implemented this approach in the IBM Threadmill post-silicon exerciser [5] to address two complex stimuli generation problems: the creation of address translation paths, and the selection of addresses for memory access management.

We achieve the high quality of the off-platform generated data by using well-established pre-silicon stimuli generation techniques. For this purpose, we employed the following pre-silicon tools: *a*) DeepTrans [4], a technology that specializes in generating sophisticated address translation paths for the functional verification of processors and *b*) Constraint satisfaction problem solver (CSP solver) [11], which takes the test-template and the system configuration as input to allocate memory regions and create address tables that are used by the exerciser during runtime. The solver outcome complies with memory requirements that are induced by the system configuration or specified by the user in the test-template. The CSP solver we used is a well-established technology and is used by IBM pre-silicon test-generators Genesys-Pro [2] and X-Gen [15]. We conducted several experiments that demonstrate the effectiveness of this method in the stimuli domains to which it was applied.

The rest of the paper is organized as follows: After discussing related work in the next section, we describe the general solution scheme for exercisers in Section 3. In Sections 4 and 5, we describe its implementation to address translation paths and memory access management (respectively) with the experiments we conducted and their results. Finally, we give our conclusions in Section 6.

2 Related Work

Post-silicon validation has been getting a lot of attention in recent years [22, 23]. However, the majority of this attention has been directed towards constructing efficient mechanisms for collecting runtime data from the chip to enhance checking and debugging capabilities [1, 12–14, 21].

More recently, other aspects of post-silicon validation have been attracting research attention. In [3, 27] an overall methodology for post-silicon is presented. Singerman et al. [28] and [8] address the issue of coverage in post-silicon.

The main topic of this paper is that of stimuli generation for post-silicon validation. Specifically, we focus on Software-Based Self-Tests, or SBSTs. In [26] authors provide a comprehensive survey on the subject. Additional recent publications on this matter can be found in [30, 31]. In [10] the authors establish the effectiveness of SBSTs for the validation of the processor's memory model

The authors of [20] and [18] propose a variety of techniques to mutate failing SBSTs in order to shorten the time between bug occurrence and its detection.

In [29], the author presents the concept of post-silicon exercisers along with a simple generation technique.

Several papers describing Threadmill, its usage and internal mechanisms have also been published. A pre- to post-silicon verification methodology is described in [3], where Threadmill is presented as the tool enabling test-plan driven post-silicon validation. Industrial experience of applying the unified methodology to POWER7® processor chip is described in [6]. Some of the Threadmill generation techniques are presented in [5]. Specifically, a technique for floating-point instruction generation is presented, which is similar in essence to the concept presented in this paper, but is confined to floating-point. Adjusting stimuli generators to leverage the unique added value of different execution platforms is described in [24].

Finally, a different mechanism for memory management is described in [7], in which the selection of addresses is performed on-platform, in a more computationally expensive manner.

3 Solution Scheme

In this section we describe our high level approach to ensuring high quality of the post-silicon validation effort while guaranteeing high silicon utilization. While we focus on the implementation of the solution in the context of exercisers, we note that the same approach can be easily applied to the general case of SBSTs. One such example is described in Section 5.

The exerciser execution process consists of two consecutive parts: the off-platform exerciser image build and the actual online run on the platform. At the offline image build, the generator code as well as some data required for the next phase are integrated into one executable image. Notably, no generation of test-cases and instructions takes place during this offline phase. The second part takes place when the image is loaded onto the platform and it starts generating the test-cases, executing them, and checking their results.

To have high-quality test-cases, the exerciser needs to use interesting inputs while generating the instructions. For example, when generating loads or stores, the generator can pick random memory locations or, preferably, it might direct the accesses toward more interesting areas, such as cache lines' or pages' borders' affinities. One of our goals, along with high-quality tests, is to ensure high utilization of the machine by maximizing the time spent in executing test-cases. Therefore, the complex task of allocating these interesting memory areas should be avoided at runtime and shifted to the offline phase, saving more cycles for test-case executions.

Our method involves preparing interesting data whose preparation is a time-consuming task at the off-platform phase and then integrating it into the executable image. This data is needed when the exerciser generates the test-cases at runtime. Such data examples are the memory intervals to select addresses from and the translation paths. Having this data ready-to-use at runtime, enables faster generation of test-cases. Also, we use well-established pre-silicon tools to ensure that the off-platform generated data is of high quality.

In cases where pseudo-random techniques are used to generate the data, such as the cases described in this paper, it is essential to rely on pseudo-random methods, and to retain the list of seeds used to create the data. This is required so that if needed, during the debug process of a fail, the same test-case, including initial values and instruction stream, can be reproduced.

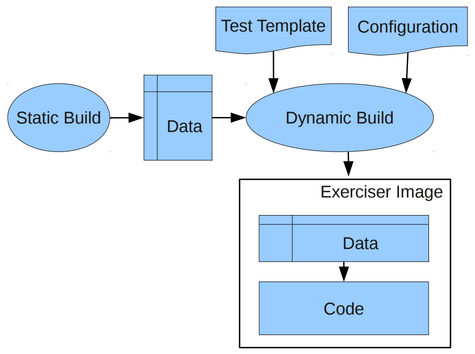

Fig. 1: Off-platform data preparation flow

We distinguish between two different stages of the off-platform data preparation: *static-build* and *dynamic-build* (see Figure 1). In the static-build stage, we generate large amounts of data by using the relevant pre-silicon tools. This data is filtered in the subsequent dynamic-build stage and are then used by the exerciser. The static-build is an independent stage that does not require specific inputs, such as the test-template and the system configuration. For example, generating large amounts of possible translation paths for the entire physical memory is independent of the number of threads in the system and of the required scenario in the test-template. Furthermore, this independence allows us to employ sophisticated pre-silicon tools that have a long runtime.

On the other hand, the dynamic-build stage takes place every time we construct an exerciser image. This stage takes, in addition to the data created at the static build stage, the test-template and the system configuration as additional inputs. This stage must therefore be sufficiently efficient, enabling the user to construct a new exerciser image in a reasonable amount of time. For example, the verification engineer may have one test-template specifying multiple inter-thread collisions and another test-template targeting page-crossing events. Each of these cases calls for different allocation of memory intervals, and therefore, these intervals must be determined at the dynamic-stage in which the test-template is available.

Overall, the dynamic-build stage has four main roles: *a*) filtering the relevant data from the static-build stage; *b*) preparing new data based on new available inputs, such as the test-template and the system configuration; *c*) organizing the data created in both stages in structures that are optimized for efficient retrieval by the exerciser during the on-platform runtime; and *d*) integrating the code and the prepared data into one executable image.

Creating the efficient data structures depends on the data type and its usage. For example, in the memory management case, we hold all the offline prepared intervals in one primary table and sort it by certain translation properties, memory ownership, and size. A memory interval can cross a page or a cache line borders. In these cases, accesses to such intervals will cause interesting events. Our method enables the exerciser to do fast retrieval of these interesting intervals by using auxiliary look-up tables per each event. Each auxiliary table contains the indices of the entries from the primary table where the relevant interesting intervals reside.

Using this method, all the exerciser has to do during runtime is to retrieve and use the pre-made data structures for generating complex test-cases that trigger interesting, meaningful events throughout the design.

4 Address Translation

Address translation is an integral part of all modern computer architectures. In addition to supporting multiple virtual spaces, it commonly plays a part in memory protection and caching mechanisms by maintaining the related properties for basic translated memory units, e.g., pages or segments. The growing demand for performance complicates the address translation and memory management mechanisms, thereby increasing the risk of bugs.

To thoroughly validate all translation-related hardware mechanisms an SBST must support a large variety of events. These coverage requirements include the ability to produce a set of valid translation paths that cover a large physical memory region. This is required to ensure that the translation mechanisms are stressed, triggering events such as TLB cast-outs and invalidations. In addition, the randomization of the numerous translation path properties is desired. In PowerPC, for example, this includes the segment size, page size, protection bits and more. Finally, the SBST must generate translation paths that activate all possible inter-thread and inter-processor memory sharing scenarios, including translating different virtual pages into the same physical page, use of the same translation table entries by different threads, and others.

The task of generating a valid translation path is a complex one. In addition to ensuring the coverage of the different properties described above, a large set of rules must be obeyed. In the PowerPC architecture, two such examples are the rule requiring that each page starts at an address naturally aligned to its size, and the rule requiring the consistency of the cachability attribute across all translation paths. Every access to an address in physical memory can go through the cache (termed *a cachable access*) or bypass it (termed *a caching-inhibited access*). This caching property is an attribute of the page mapped to the given physical address. Therefore, for every accessible address in physical memory, the caching property must be consistent across all pages mapped to that address.

	Small	Medium	Large	Huge
Cacheable	3230	3202	580	37
Caching-inhibited	386	380	62	36

Table 1: Distribution of page sizes and caching properties

To address the problem of translation path generation at the pre-silicon verification stage a tool called DeepTrans [4] is used. DeepTrans contains rich built-in testing knowledge enabling the generation of interesting translation paths with or without specific user requests.

We propose to address the challenge of generating valid and interesting translation paths through a solution partitioned among the SBST run phases, namely, the static-build stage, the dynamic-build stage, and the on-platform runtime stage.

At the static-build stage we leverage DeepTrans [4] to construct a large set of address translation paths for the entire physical memory. We iteratively activate DeepTrans to generate, at each iteration, a new translation path. Each of these translation paths is generated under different constraints. These constraints may request, for example, that the next translation path is generated to create a $4KB$ page access in a caching-inhibited way, leaving many other attributes for DeepTrans to randomize. By using these constraints we ensure that every location in physical memory is accessible in every possible mode, through more than one path. As DeepTrans has all PowerPC translation rules modeled in it, all generated paths are legal and consistent with each other.

To evaluate the quality of our approach we ran DeepTrans to generate $7,913$ translation paths covering a physical space of $8GB$. Overall, 20 hours were required to generate such a set of paths on a single Intel Xeon® Linux server running at $2.4GHz$ with $16GB$ of RAM.

Table 1 depicts the distribution of page sizes and the caching properties over the generated set of translation paths. We strongly bias DeepTrans towards generating pages that are accessed through the cache, as we wish to stress the entire memory hierarchy. In addition, we bias DeepTrans towards generating small or medium pages, as such pages provide us more freedom in triggering some interesting events. For example, a page crossing access, i.e., a memory access that spans more than one page, can only occur at the boundaries of pages. Partitioning memory to small pages provides a larger set of addresses where a page crossing event can be triggered.

A *translation collision* event occurs when same memory location is accessed using different translation paths (either from the same or different hardware threads). These accesses may vary in one (or more) of the different translation attributes. Recall that some attributes, such as the caching property, must be consistent across all paths accessing the same physical address.

	Small	Medium	Large
Medium	203		
Large	97	603	
Huge	53	66	22

Table 2: Different page size collisions

We turn to evaluate the quality of the set of paths generated by DeepTrans with respect to the potential of triggering a translation collision event[1]. Specifically, we evaluate the potential of triggering a translation collision where the translation paths differ in page size. We divide the available page sizes in the PowerPC architecture into 4 categories: small ($4KB$ and $16KB$), medium ($1MB$), large ($8MB$), and huge ($8GB$). We measure the absolute number of cases when a physical page covered by a translation path with a larger page size contains a physical page covered by another translation path with a smaller page size. Results are shown in Table 2. The numbers inside the table cells show the absolute number of these cases; the corresponding row and column show the corresponding page sizes. As can be observed, we generated all possible collision combinations. The bias toward the collision with large pages represents the internal DeepTrans expert knowledge that these collisions are more valuable than the collisions with huge pages. For example, the number of collisions between large pages and medium size pages (603) is almost ten times higher than the number of collisions between huge pages and medium size pages (66).

At the dynamic-build stage the test-template and system configuration are accessible, in addition to the complete set of pre-generated translation paths. We use this data to pick and choose translations paths that best match the intent expressed in the test-template. For example, if the test-template targets page crossing accesses between caching-inhibited pages, we bias our path selection to ensure a variety of matching paths are incorporated in the exerciser image.

In addition to the selection of paths, we also arrange the data in a way that will facilitate an efficient access during runtime. For example, in our Threadmill implementation of this approach, translation paths are arranged in two tables. The first table is sorted by the test-case's runtime mode and the physical address – this is required to enable the memory management component (which selects physical addresses) to effectively map them to virtual addresses. The second table is sorted by the virtual address and is used by the translation exception handlers in order to facilitate an effective installation of the translation path.

At runtime, the memory management component and the translation exception handlers access the tables to extract the required data as explained above. Overall, about 9 seconds are needed for DeepTrans to generate a single translation path. Doing this on the silicon platform would sharply drop its utilization, and

[1] The set of translation paths only describes the potential to trigger a translation collision event; it is up to the memory management mechanism to create such events through an intelligent selection of addresses to access within a test-case.

the majority of the run time would be spent building translation paths instead of actually executing test-cases. In addition, adding such complex algorithms to the exerciser would make it far more complex and difficult to debug.

Using this method, a diverse set of test-templates targeting different aspects of the translation mechanisms, coupled with several sets of pre-generated translation paths, enable us to address coverage holes and achieve an aggregated high coverage.

5 Memory Management

Modern high-end multi-threaded systems rely on *weak* consistency memory models [9,16] that make it easier to implement performance boosting mechanisms such as caches, out-of-order, and speculative executions. Implementations of these weak consistency models are highly error-prone and hard to verify due to the vast test space and their distributed nature.

One desired attribute of SBSTs targeting the validation of the memory hierarchy is the ability to generate *collision events*. A collision event occurs when two (or more) hardware threads access the same memory location. Modern processors manage data transition in cache line granularity, that is, regardless of the size of the program access, the processor fetches data from memory in chunks of fixed size. Therefore, to stress the memory hierarchy, it is often enough to have the hardware threads access different locations within the cache line (termed *false sharing*) rather than the exact same location (termed *true sharing*).

The generation of memory access collisions must take into account the checking method. Threadmill employs a technique called *multi-pass consistency checking* [5]. In this technique, a test-case is run multiple times with the same resource initializations and verified to ensure that the same final values are produced each time. 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 differ for different executions of the same multi-threaded test-case. Similarly, a write-read collision may result in different values in the target register of the read. For this reason, Threadmill checks neither the memory used for write-write collisions nor the registers used in write-read collisions. It still makes sense to generate these unchecked collision events because they stress the hardware and may cause failures that can be observed by other means (e.g., built-in hardware checkers).

Threadmill supports this by allocating, to each hardware thread, a number of *owned* intervals. Only the "owner" thread is allowed to write to these intervals. Additionally, all threads share *read-only* and *unowned* intervals. The unowned intervals can be written to and read by all threads but are left unchecked, while the read-only intervals are only read. This ownership scheme allows Threadmill to produce different types of true-sharing and false-sharing collision events while maintaining its checking method.

A user can explicitly direct certain memory accesses, for example, to target the same cache congruence class. This can be specified in the test-template by defining a *mask*, which is a bit-vector with don't-care (X) values for some of its bits. Each mask represents the set of addresses that can be obtained by determining don't-care bits. For example, the mask $8b11XXXX00$ represents all the 8-bit addresses that start with two 1s and end with two 0s.

Threadmill strives to place the intervals at interesting locations to increase the test-case quality. A memory location is considered *interesting* if accessing it stresses the design. Such locations include a) cache line or page/segment crossing; b) memory having certain attributes, for example, non-cacheable memory or memory obeying different consistency rules; and c) various memory affinity, such as memory located on a different chip.

We employ the CSP solver, feeding it with our requirements to produce the memory intervals that the exerciser will use during runtime. Some inputs to the solver, such as the test-template and the system configuration, are only available in the dynamic-build stage, thus making executing the solver in the static-build stage impossible. Our CSP technology proved to be time-efficient, taking less than 10 seconds for 48 threads, which is a typical bring-up configuration that satisfies the build time requirements of the dynamic-build stage. The resulting set of memory intervals includes: memory for the code and data areas of the exerciser application, memory for test-cases that will be generated during runtime, and the memory accessed by the generated load/store instructions. The intervals are embedded into the exerciser and organized such that they can be efficiently retrieved by it during runtime.

As in the case of the address translation paths, conducting such computations would drop silicon utilization dramatically. Our solution calls for moving this computation off to the dynamic-build stage. At runtime, the generator can randomly select one of the available entries from the table to find a memory location to access (details below).

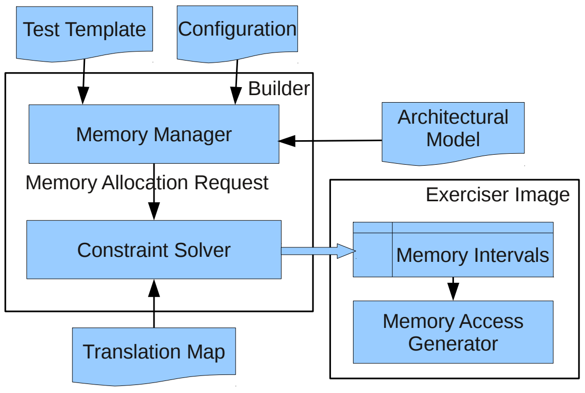

Fig. 2: Memory management architecture

The high-level architecture of Threadmill's memory management component is depicted in Figure 2. This component consists of two sub-components: a builder application and a memory access generator. The builder application runs during the dynamic-build stage and allocates all the memory intervals. The chosen intervals are incorporated into the exerciser image as tables organized for quick retrieval. The memory access generator component executes during runtime. It retrieves a matching interval address from the tables, based on the randomly generated memory access. Note that the tables are built to ensure sufficient variability in the address selection.

The builder retrieves the user requests from the test-template. It also retrieves the system topology, the number of threads, and the available memories from the system configuration. The builder then accordingly creates memory allocation requests and feeds them to the CSP solver. The memory allocation requests consist of instruction stream locations (test-cases), user-defined memory allocations, and allocations for random loads and stores (collision areas). The CSP solver also accepts a memory map produced by the address translation component (see Section 4). The map describes page and segment boundaries as well as page attributes, e.g., whether a page is cacheable or non-cacheable.

The CSP solver represents each memory allocation request as a pair of CSP variables: interval start and interval length. There are two common mandatory (also called *hard*) constraints: *a*) all intervals are disjoint and *b*) all intervals reside in the available memory space. We also add specific hard constraints for each memory allocation type. For example, for a user-defined memory allocation request, the CSP solver must allocate an interval of the required size, starting at an address that adheres to the specified mask. For random loads and stores, the CSP solver must allocate the required number of intervals for each ownership type (read-only, owned, and unowned). We also specify the size distribution of the intervals, as well as the minimal number of cacheable and non-cacheable intervals for random memory accesses, enabling Threadmill to find a matching interval for any possible memory access. We allocate several intervals for each request to ensure sufficient variability during runtime.

In addition to hard constraints, the CSP solver is also passed a number of non-mandatory (also called *soft*) constraints to enable the generation of high quality test-cases. These soft constraints are satisfied on a "best effort" basis and used to direct interval allocation to interesting areas such as multiple intervals residing within the same cache line. The builder creates data structures containing the allocated addresses and incorporates them into the exerciser. The instruction stream allocations and user-defined intervals are put into simple arrays. The collision area, however, requires some organization to facilitate quick retrieval during runtime. This is further explained below.

During runtime, the memory access generator first decides on the generated instruction and then decides on the memory access address (see Figure 3).

With an instruction at hand, we know whether the memory access is cacheable or non-cacheable, load or store. Cacheable and non-cacheable memory areas are mutually exclusive, thus we divide the collision area table into two parts. We

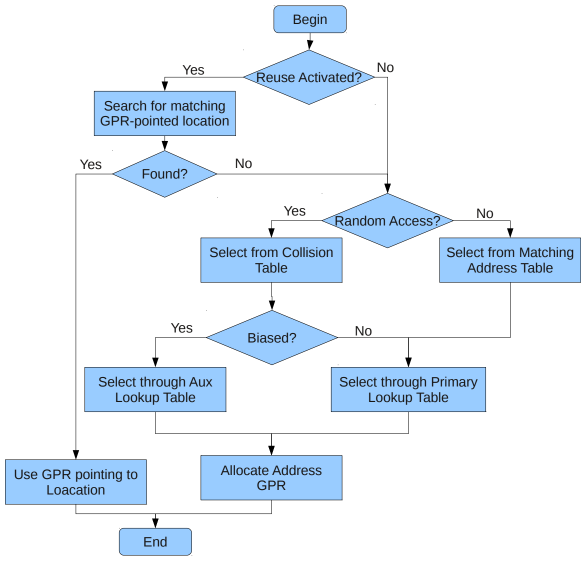
Begin
Reuse Activated?
Yes
No
Search for matching GPR-pointed location
Found?
Yes
No
Random Access?
Yes
No
Select from Collision Table
Select from Matching Address Table
Biased?
Yes
No
Select through Aux Lookup Table
Select through Primary Lookup Table
Use GPR pointing to Loacation
Allocate Address GPR
End

Fig. 3: Random access flowchart

also group all the memory intervals according to their ownership. Based on whether our memory access is load or store, checked or unchecked, we can choose the matching ownership regions and randomly select from them. We construct a primary look-up table for each thread. This table allows for fast and simple random choice (by just a single operation), while maintaining uniform distribution amongst entries. This look-up table contains indices into the main collision area table, grouped by access type (see Figure 4). Organizing the collision area tables in this way enables fast and efficient data retrieval, resulting in high machine utilization.

We also build auxiliary look-up tables per event. These table are used to support fast retrieval of addresses when the generation of a specific event is desired. These tables contain indices from the primary table that support the required event. Consider, for example, the case of a page-crossing access event. Only a subset of the chosen addresses may support this event, as it requires for the memory interval to start "near" the end of one page, and be long enough to cross into the next. All relevant memory intervals supporting this event are identified at the dynamic-build stage, and their indices are places in the page-crossing look-up table. The auxiliary look-up tables are sorted by index number, and hence, by construction, maintain the same sorting attributes of the primary table.

To demonstrate the effectiveness of our scheme, we conducted two experiments. For both experiments, we ran Threadmill on 3 configurations: 2, 4, and 8 threads, all with $8GB$ of memory and 10 intervals allocated per ownership. We generated 50 load/store instructions per thread and measured the number of collision events generated for the checked accesses. A collision event is defined as a pair

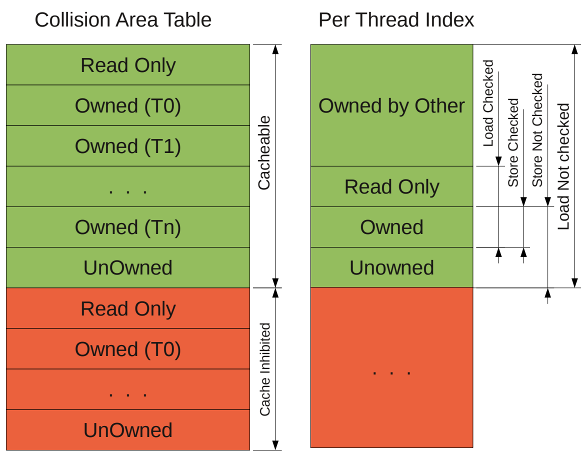

Fig. 4: Collision area tables

Threads	Rd-Rd	Wr-Wr	Rd-Wr	Total Intervals
2	17.2	0	0	40
4	74.6	0	0	60
8	508.6	0	0	100

(a) Without collision bias

Threads	Rd-Rd	Wr-Wr	Rd-Wr	Total Intervals
2	35.6	33.6	53.8	40
4	147.6	80.8	156.2	60
8	714.2	207.4	420	100

(b) With collision bias

Table 3: Average number of collisions

of accesses by different threads that target (possibly different) locations in the same cache line. For example, suppose thread 0 loads twice from some cache line and thread 1 stores three times to the same cache line. This is counted as six read-write collision events. The results of the experiments are presented in Table 3, in which every row gives the average of five different runs. Every column except the last one shows collisions between different types of memory accesses. The last column shows the total number of intervals allocated. For example, for 8 thread configurations we allocated 100 intervals and 714.2 collisions occurred between Read and 207.4 between Write accesses of different threads, as shown in Table 3b.

In the first experiment, the intervals were allocated randomly and uniformly distributed within the available memory. As the experiment results show, the probability of generating colliding memory accesses for randomly allocated intervals within a large memory space is negligible. Note that we still obtain Read-Read collisions since read-only intervals are shared among all threads.

In the second experiment, we used our CSP solver to bias interval allocation and increase the collision rate. We observe a considerable collision rate for all collision types, as shown in Table 3b.

While the above describes an implementation specific to Threadmill, a similar approach can be implemented to improve the quality of other tools. Consider, for example, the *Litmus* system described in [10]. The developers of the Litmus system acknowledge the need for multitude of address to efficiently utilize the hardware platform. To address this need, they opt to allocate arrays in memory (instead of a single address). However, compilers allocate arrays in contiguous memory, that is, all addresses reside consecutively in memory. Alternatively, applying our method and allocating multiple locations (and arranging their addresses in an array) would require a negligible addition to compilation time, and just one additional pointer traversal at runtime to reach the desired address.

6 Conclusions

We introduced a novel method that bridges two conflicting requirement, namely, to conduct high quality functional validation while ensuring high silicon utilization. Our method calls for shifting the preparation of complex data from the SBST's runtime to the offline phase that takes place before the software test is compiled and loaded onto the silicon platform.

We implemented this method in Threadmill and applied it to two domains: address translation path generation and memory access management. We used well-established pre-silicon technologies to ensure the quality of the generated data. Our results indicate that we are able to guarantee a high level of coverage, while keeping the on-platform complexity low. The addition of this method did not change any of Threadmill's attributes, specifically, its coverage (i.e., the type of test-cases that can be generated by Threadmill) and the ability to reproduce an exerciser image that re-generates the same test-cases.

While our results indicate a good balance between off-platform pre-computation and on-platform data usage, many additional research questions remain. One such question is the applicability of our proposed scheme when the target platform is an accelerator or emulator, as opposed to real silicon. Since accelerators/emulators are significantly slower than silicon, but significantly faster than software simulation, we believe the tradeoff might be different.

References

1. M. Abramovici, P. Bradley, K. N. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *DAC*, pages 7–12, 2006.
2. 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 & Test of Computers*, 21(2):84–93, 2004.
3. A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann. A unified methodology for pre-silicon verification and post-silicon validation. In *DATE*, pages 1590–1595, 2011.
4. A. Adir, R. Emek, Y. Katz, and A. Koyfman. Deeptrans - a model-based approach to functional verification of address translation mechanisms. In *MTV*, pages 3–6, 2003.
5. A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv. Threadmill: a post-silicon exerciser for multi-threaded processors. In *DAC*, pages 860–865, 2011.
6. A. Adir, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann. Leveraging pre-silicon verification resources for the post-silicon validation of the IBM POWER7 processor. In *DAC*, pages 569–574, 2011.
7. A. Adir, A. Nahir, and A. Ziv. Concurrent generation of concurrent programs for post-silicon validation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(8):1297–1302, 2012.
8. A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *HVC*, 2010.
9. S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.
10. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, pages 41–44, 2011.
11. 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.
12. K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *HPCA*, pages 415–426, 2008.
13. 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.
14. A. Deorio, J. Li, and V. Bertacco. Bridging pre- and post-silicon debugging with BiPeD. In *ICCAD (to appear)*, November 2012.
15. R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. X-gen: a random test-case generator for systems and socs. In *HLDVT*, pages 145–150, 2002.
16. K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.
17. R. Gray. Post-silicon validation experience: History, trends, and challenges. In *GSRC Workshop on Post-Si Validation*, June 2008.
18. T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra. QED: Quick error detection tests for effective post-silicon validation. In *ITC*, pages 154–163, 2010.

19. J. Keshava, N. Hakim, and C. Prudvi. Post-silicon validation challenges: how EDA and academia can help. In *DAC*, DAC '10, pages 3–7. ACM, 2010.
20. D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra. Quick detection of difficult bugs for effective post-silicon validation. In *DAC*, pages 561–566, 2012.
21. S. Mitra, D. Lin, N. Hakim, and D. S. Gardner. Bug localization techniques for effective post-silicon validation. In *ASP-DAC*, page 291, 2012.
22. S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *DAC*, pages 12–17, 2010.
23. 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 *DAC*, pages 94–95, 2010.
24. A. Nahir, A. Ziv, and S. Panda. Optimizing test-generation to the execution platform. In *ASP-DAC*, pages 304–309, 2012.
25. P. Patra. On the cusp of a validation wall. *IEEE Design and Test of Computers*, 24:193–196, 2007.
26. M. Psarakis, D. Gizopoulos, E. E. Sánchez, and M. S. Reorda. Microprocessor software-based self-testing. *IEEE Design & Test of Computers*, 27(3):4–19, 2010.
27. H. G. Rotithor. Postsilicon validation methodology for microprocessors. *IEEE Design & Test of Computers*, 17(4):77–88, 2000.
28. E. Singerman, Y. Abarbanel, and S. Baartmans. Transaction based pre-to-post silicon validation. In *DAC*, pages 564–568, 2011.
29. J. Storm. Random test generators for microprocessor design validation. http://www.oracle.com/technetwork/systems/opensparc/53-rand-test-gen-validation-1530392.pdf, 2006. Accessed: 2013-09-01.
30. G. Theodorou, S. Chatzopoulos, N. Kranitis, A. M. Paschalis, and D. Gizopoulos. A software-based self-test methodology for on-line testing of data tlbs. In *European Test Symposium*, 2012.
31. G. Theodorou, N. Kranitis, A. M. Paschalis, and D. Gizopoulos. Software-based self test methodology for on-line testing of l1 caches in multithreaded multicore architectures. *IEEE Trans. VLSI Syst.*, 21(4):786–790, 2013.