Maximizing Power State Cross Coverage in Firmware-based Power Management*

Vladimir Herdt[1] Hoang M. Le[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de

Abstract— Virtual Prototypes (VPs) are becoming increasingly attractive for the early analysis of SoC power management, which is nowadays mostly implemented in firmware (FW). Power and timing constraints can be monitored and validated by executing a set of test-cases in a power-aware FW/VP co-simulation. In this context, cross coverage of power states is an effective but challenging quality metric. This paper proposes a novel coverage-driven approach to automatically generate test-cases maximizing this cross coverage. In particular, we integrate a coverage-loop that successively refines the generation process based on previous results. We demonstrate our approach on a LEON3-based VP.

I. Introduction

Stringent requirements on power consumption and performance have been putting the emphasis on power optimization already in early design steps at the system level, as both software (SW) and hardware (HW) have a significant impact on the overall power consumption. At this level, one of the main opportunities for power saving is the development of efficient *Power Management (PM) strategies*. An efficient strategy will put each component into an appropriate *power state*, so that the system will only consume "just enough" power to meet the deadline of the current workload. Due to the flexibility in adapting for different target applications, the global PM strategy is implemented in FW in most modern SoCs.

Such a FW-based PM solution must be thoroughly validated before deployment to ensure that it will perform as expected, i.e. neither the power budget is exceeded or the performance constraint is violated. As an example, too aggressive power-down might cause delay in processing and affect functional correctness. While a validation on the production-level SW and the target HW platform is unavoidable, one needs to start much earlier. The reason is that detecting a major power-related HW/SW issue after the RTL is already written is too late and fixing it will be very costly.

The recent advances at the *Electronic System Level* (ESL) have laid the foundation for early validation of FW-based PM. On one hand, SystemC *Virtual Prototypes* (VPs) enable very fast HW/SW co-simulation [1, 2, 3]. On the other hand, emerging ESL power modeling and estimation techniques (see e.g. [4, 5, 6]) allow such high-level co-simulation to be reasonably accurate wrt. power and timing. Building on this foundation, *constrained random* (CR) techniques [7, 8], being previously predominantly used for functional verification, have been lifted to the validation task of FW-based PM strategies [9]. Instead of real SW applications, system-level workload scenarios can be described by a set of constraints and then synthetic SW workloads, in the following we refer to them also as test-cases, can be generated in a fully automated manner. Each such workload scenario corresponds to a system-level use-case with an intended power consumption and performance profile.

Although the constraint-based description enables automated generation of a large number of different test-cases (corresponding to SW workloads), hence reducing the risk of missing a corner case, a coverage metric to objectively measure the quality aspect as well as to guide the generation of scenarios is still an important missing piece. At the very least, it is mandatory that all power states of each component are comprehensively exercised by the generated test-cases. Recent experience from the industry [10] makes a case for using stronger metrics. The paper argues that the power states from different components or power domains are not necessarily independent. This also applies to our context of FW-based PM, since this global management scheme can change the power state of several components simultaneously according to the implemented strategy. Therefore, an appropriate coverage metric must account for all possible combinations of these interdependent power states. The cross coverage of power states is such a metric.

This paper proposes a novel coverage-driven validation approach for FW-based PM. The main contribution is a feedback-directed workload generation algorithm that generates test-cases in an automated manner in order to maximize the cross coverage of power states. Our approach works in two phases: first a bootstrap phase is performed to obtain preliminary coverage information based on randomly generated test-cases and then a coverage-loop phase to close the remaining coverage gaps. The coverage-loop works in (two) different generation modes and integrates a refinement loop to guide the test-case generation process. We demonstrate the applicability and efficiency of our approach using the open-source SoCRocket VP [11] and four different PM strategies implemented in FW.

The rest of this paper is organized as follows: We start by reviewing related work. Then we provide relevant background information in Section II. Next, in Section III, we present our approach on a coverage-driven maximization of power state cross coverage. In Section IV we present the results of the case study using the SoCRocket VP with four different PM strategies. We conclude with a discussion on limitations and future work.

Related Work We are not aware of any other coverage-driven validation approach for FW-based PM. A feedback-directed algorithm for maximizing cross coverage of power states is therefore, to the best of our knowledge, completely novel.

In the area of functional verification, automated coverage-driven verification has been one of the most important research topics and thus

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and by the University of Bremen's Central Research Development Fund and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

ASPDAC '19, January 21-24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01...$15.00
https://doi.org/10.1145/3287624.3287631

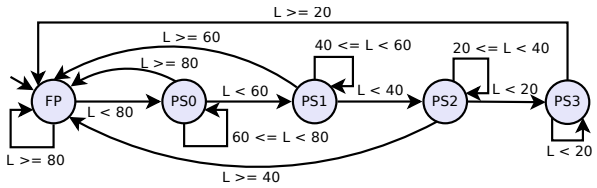

Fig. 1: Example power FSM of a component implementing an *on-demand* PM strategy. L denotes the duty cycle (i.e. load) of the component in the last period (time between FW update cycles).

received much attention. Current state-of-the-art approaches, pioneered by IBM Research [12], are also based on a feedback-directed loop from achieved coverage to the scenario generator. They often integrate machine learning techniques, such as Bayesian networks or data mining, to infer the assumed relationship between coverage and scenario generation directives. We refer the reader to an excellent survey [13] on this topic for more details. These approaches are powerful but generic. The specific context of FW-based PM allows us to develop a specialized, much simpler feedback-directed algorithm. It would be very interesting to compare the performance between the proposed approach and those generic solutions. Unfortunately, they are not publicly available. A more recent and very promising coverage closure approach is based on assertion mining[14]. However, it requires an output-directed notion of coverage that is not directly transferable to our context.

Formal techniques have been shown to be feasible for the *verification* of FW-based PM, see e.g. [15]. The problem being solved is however completely different from validation. Such approach aims to ensure the correctness of a concrete HW/SW implementation of a PM strategy (and thus can only be applied very late in the design flow), while we want to evaluate the effectiveness/soundness of this strategy from the system-level perspective.

II. Preliminaries

In this section we first briefly describe the kind of systems that we consider in this work. In particular we focus on a duty cycle based PM which is implemented in FW and executed on a VP. This setting represents an important class of systems. Then in Section B we introduce the concept of a cross power FSM (finite state machine).

A. Duty Cycle based Power Management in Firmware

In the VP every component (e.g. CPU, bus system, memory, interrupt controller, etc.) is associated with a set of power states and defines which transitions between the power states are valid. Besides the obligatory full power (FP) state, every component is typically associated with various power save (PS) states (e.g. PS0, PS1, etc) with PS0 being the least power saving state in the hierarchy. The VP itself does not provide any logic to initiate a transition between power states. The power management (PM) strategy is completely implemented in firmware (FW).

Therefore the HW provides a *power interface unit* (PIU) which acts as power interface between the HW and FW. The PIU provides memory mapped addresses to the FW, which the FW can write and read. In particular the PIU has two tasks:

1. Decode power state change commands written by the FW and then initiate the power transition of the addressed component.
2. Collect and provide HW performance characteristics to the FW, which can be read through the memory mapped addresses. In particular the idle and active times, i.e. the duty cycle, of the HW components. The duty cycle (or load) denotes the percentage of time a component has been active in a given period of time.

The PIU will trigger periodic interrupts that will initiate an update cycle in the FW (e.g. every 50ms). Essentially, the interrupt handler in the FW will read the collected duty cycles and based on this information decide which (if any) power state transitions to perform. For example Fig. 1 shows a specification of an *on-demand* PM strategy for a CPU (or some other component). The states of the FSM denote power states and the directed edges that a transition is possible. The edges are annoted with the duty cycle ranges which will trigger the transition. For example if the CPU is in state PS1 and a duty cycle (i.e. load) $L \geq 60$ is observed, i.e. the CPU has been active between 60% and 100% in the last period, then the FW will initiate a transition to the full power (FP) state.

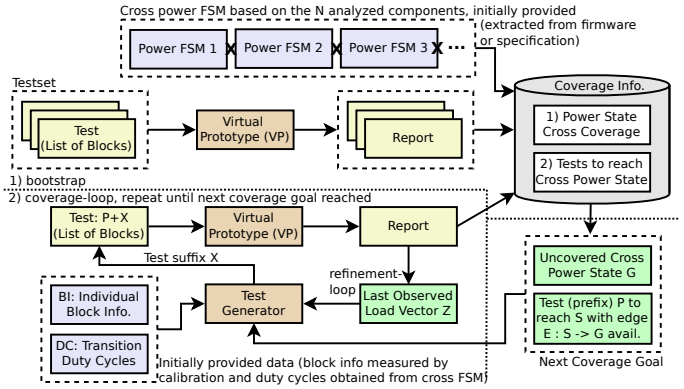

Fig. 2: Overview of our approach

B. Cross Power FSM

A power FSM, as introduced in the previous section, can be naturally extended to a cross power FSM. Essentially, the cross FSM is a product machine of the N individual power FSMs. A transition between two states $S = (a_1, ..., a_N)$ and $G = (b_1, ..., b_N)$ in the cross FSM is possible, if the transitions (a_1, b_1), ..., (a_N, b_N) are defined in the individual power FSMs, respectively (where a_1,...,a_n and b_1,...,b_N are power states from the corresponding individual FSMs). The duty cycles annotated to a cross transition are simply a combination of the annotations of the individual FSMs transitions. For example consider a cross FSM, which is a combination of three FSMs, all three as defined in Fig. 1. This cross FSM has an edge from state $(PS0, PS1, FP)$ to state $(PS0, PS2, PS0)$ annotated with the load expression ($60 \leq L1 < 80$ and $L2 < 60$ and $L3 < 80$) where L1, L2 and L3 are the load (i.e. duty cycle) annotations from the three individual FSMs.

III. Maximizing Power State Cross Coverage

As already mentioned, we assume that the PM strategy is implemented in FW and updated periodically based on the duty cycles of the components in the system. We have described such systems in more detail in the preliminary Section A.

First we select a number of components from the system and create the cross power FSM based on the individual power FSMs (see Section B for more details on this preliminary step). The individual power FSMs where the (directed) edges are annotated with transition duty cycles (e.g. as shown in Fig. 1) can be obtained from the specification or extracted from the FW for each component. Then, we generate test-cases that will maximize the coverage of the cross FSM states. We describe our proposed approach in the following.

A. Overview

An overview of our approach is shown in Fig. 2. Our approach works in two phases: first a bootstrap phase (upper part of Fig. 2) is executed to obtain preliminary coverage information and then a coverage-loop phase to close the remaining coverage gaps (lower part of Fig. 2). We keep track

of relevant coverage information in a shared data structure (right side of Fig. 2). In the following we describe our approach in more detail.

In the *bootstrap phase* a set of test-cases one after another is linked with the FW, then cross-compiled and executed on a power-aware VP. These test-cases can be obtained in various ways, for example created by a testing engineer or using (constrained) random generation techniques. We assume that each test-case consist of a list of blocks (i.e. the SW workload). A block is a list of instructions, potentially including (bounded) loops. Each block corresponds to a specific instruction execution profile (e.g. memory, arithmetic, sleep, etc.) and runs only for a short amount of time compared to the FW-based PM update cycle interval. The number of blocks can be randomly chosen, but it should be long enough to trigger (preferably multiple) PM update cycles to obtain useful coverage information. For each execution an (execution) report is obtained. The report contains various informations about the test-case execution, including the execution time of each block and the power state changes of every component in the system during the PM update cycles.

A shared data structure stores the relevant coverage information from the reports between all execution runs. Essentially, these are two pieces of information: 1) Based on the observed power state changes in the report, the visited cross power states are marked. This information is used to select the next uncovered goal state G. 2) A mapping W from cross power state S to list of blocks B (i.e. test-case or prefix of a test-case) is updated, such that the execution of B will lead to the cross power state S. This information is used to generate a prefix of blocks to reach a specific power state.

The *coverage-loop phase* starts after all test-cases of the bootstrap phase have been executed to close the remaining coverage gap. It will consider all edges `E = S -> G` from the cross FSM, where the start S is covered and the goal G is not, one after another. Based on the mapping W from the shared coverage data, a prefix P of blocks is available whose execution will reach the state S. Thus, it is only necessary to generate a suffix X in order to hit the (cross) power transition `S -> G`. Therefore, the test generator first extracts the goal load from the cross FSM. The goal load is an interval vector (GLIV), where each interval denotes the expected duty cycle for every component in the cross FSM to apply the cross power transition `S -> G`. For example consider a cross FSM for three components, each using the Fig. 1 power FSM individually, with the start state S=(PS0, PS1, FP) and the goal state G=(PS0, PS2, PS0). Then the goal load (cuboid) is defined as ($60 \leq L1 < 80$ and $L2 < 60$ and $L3 < 80$) where L1, L2 and L3 are the duty cycles (i.e. loads) of the three components. Based on the GLIV (i.e. goal load), the suffix X of blocks (long enough to reach at least one PM update cycle in order to have an effect) is generated. As already mentioned, this suffix X should consist of blocks such that execution of (P + X) will trigger the cross power transition from S to G, i.e. reach the expected goal load in the next PM update cycle. Fig. 3 shows the principle[1]. After executing the test-case (P + X) there are two possibilities:

1. The cross state G has been covered. In this case simply proceed to the next uncovered cross state.
2. Otherwise, the suffix X needs to be refined. Refinement is based on the information that has been collected during execution of (P + X). In particular the load vector Z, which contains the observed duty cycle for each component (obtained from the first update cycle when executing X, which is U4 in Fig. 3). Refinement and re-execution will iterate until G is covered or the test generator gives up on refinement, e.g. because the maximum number of refinement steps is reached. In case refinement is not possible, the goal state G might still be covered later, because in the cross power FSM there can be multiple edges with different start states S that reach G.

[1]The execution of the last block of P can still reach into the execution period of X (i.e. the beginning of the interval between U3 and U4 in Fig. 3). However, by keeping the blocks short compared to the length of the periods between update cycles, this last block of P has only negligible influence on the overall execution of X.

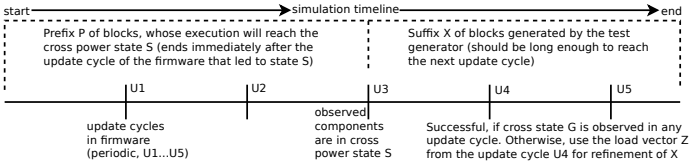

Fig. 3: Conceptual overview of the test generation and execution process in the coverage-loop

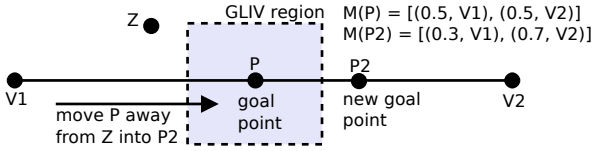

Fig. 4: Abstract example demonstrating the line-mode refinement approach in a two dimensional space.

We provide more information on the test generation process in the following.

B. Coverage-loop

The coverage-loop test generator starts with a goal load interval vector (GLIV), as described in the previous overview section, and generates a suffix of blocks X. The test generation is based on mixing blocks to obtain specific load intervals for the components within the system. Therefore, a block calibration is performed once before the coverage-loop starts to obtain *individual block information* for every block. Therefore, every (instruction) block is executed individually on the virtual prototype (VP) for a fixed number of times (long enough to trigger an update cycle in FW) while keeping the system in full power mode. By doing so a concrete load vector (i.e. duty cycles of each component of the cross FSM) and the (average) execution time of one individual block is obtained and stored. We perform the calibration only in full power mode to avoid the state space explosion of having to consider the exponential many (in the number of components in the system) different power state configurations of the system. Please note, this calibration has only to be done once. In Section IV we present concrete blocks and calibration results for our case study.

Based on these individual block information obtained from calibration, the test generator works iteratively through multiple different modes, starting from simple to more elaborate ones. In this work we consider the *point-mode* and *line-mode* generation.

First the *point-mode* generation is applied. Therefore, every individual block load vector V (obtained through initial calibration) is checked against the goal load interval vector (GLIV). If V is contained in GLIV (for example with three components V would be a point and GLIV a cuboid) the corresponding block B of V matches the goal load. In this case the test generator will generate a suffix X consisting only of B blocks. The mixing vector is defined as [(1.0, B)], i.e. only a single block is used with full weight (i.e. a factor of 1.0). The expectation is that the observed load vector after execution of X will match V and hence cover the GLIV. In case the execution does not match, the next individual block is considered. No refinement is performed in point-mode, since only a single block is used. In case the GLIV cannot be matched using *point-mode*, the generator proceeds to *line-mode*.

In *line-mode* the generator will consider all lines obtained by combination of two block load vectors. For every such (finite) line denoted by two (end-)points L=(V1, V2) the intersection with the GLIV is tested. If there is an intersection, the closest point P on the line L to the center point of the GLIV is computed. The mixing vector M is computed based on the (inverse) distance of P to the edge points V1 and V2 of L. Let d1 and d2 be the distances of P to V1 and V2 with B1 and B2 being the blocks associated with V1 and V2, respectively. Then M is defined as M=[(d2 / (d1 + d2), B1), ((d1 / (d1 + d2)), B2)]. The division ensures that the factors

TABLE I: Example that demonstrates the generation of blocks from a mixing vector by interlacing them.

step		0	1	2	3	4	5	6	7
budgets	B	0	0.5	0.3	0.1	-0.1	0.4	0.2	0
	C	0	0.6	0.5	0.4	0.3	0.2	0.1	0
blocks			A	A	A	AB	A	A	ABC

sum up to 1.0, i.e. are normalized. The expectation is that the observed load vector Z after execution of X will match the point P and hence cover the GLIV. In case the execution does not match, a refinement approach is started. The refinement approach will modify the block weights of the mixing vector M. Essentially, it will move P in the direction of either V1 or V2, away from the observed vector Z. Fig. 4 shows the principle. The new goal vector P2 has a stronger influence of V2 (hence its weight factor has increased from 0.5 to 0.7) and thus we expect that the next observed load vector will move towards the original goal vector P and thus be enclosed by the GLIV.

Further modes can be defined, for example using planes or tetrahedra in combination with barycentric coordinates to allow for a more flexible mixing of blocks. However, our experimental evaluation in this paper indicates that using the line-mode can already be sufficient to obtain (close to) maximal power state cross coverage. Therefore, we leave it for future work to implement and evaluate more sophisticated mixing modes and refinement procedures. In the following we describe how to transform a mixing vector to a concrete list of blocks (i.e. the suffix X).

C. Final Test Generation

Given a mixing vector of blocks, for example M=[(0.7, A), (0.2, B), (0.1, C)], the final task is the generation of the actual suffix X of blocks. This happens in two steps:

First the factors are re-scaled by the average individual block execution time, which has been obtained through the initial calibration. The reason is that some blocks might execute for a (significantly) longer time than others, thus it is necessary to divide the factors by the execution time to keep the proportion of the blocks in the final suffix X intact. After re-scaling, the factors are normalized again to sum up to 1.0 and passed to the next step.

Before explaining the next step, please note that the block execution time can deviate at runtime compared to the calibration result. The reason is that calibration happens in full power mode only, but during execution a PM strategy is active, and caching effects can have an impact on block execution when blocks are mixed. Further, the overall block length need to be long enough to ensure that an PM update cycle is triggered in the FW (otherwise no power state transition happens). Therefore, the number of blocks generated as suffix should be conservatively (over-)approximated.

However, in doing so, one has to be careful when generating blocks. For example consider the mixing vector M=[(0.7, A), (0.2, B), (0.1, C)] and assume that 500 blocks should be generated. With an update cycle interval of for example 50ms and a minimal block execution time of 0.2ms in full power mode this is a valid approximation. But simply generating 350 (0.7*500) A, 100 (0.3*500) B and 50 (0.1*500) C blocks one after another will not work as expected. The reason is that the execution time of 350 A may already exceed the 50ms update interval, rendering the mixing invalid (because the blocks B and C will not influence the duty cycles in this update interval).

Therefore, we employ an algorithm to interlace all blocks from the beginning. Table I shows the result of the algorithm for the first seven steps. The first row lists the step number (zero is the initialization). The second and third rows shows the budgets for the blocks B and C. The fourth row shows which blocks are generated in the corresponding step. The algorithm works as follows: First it obtains the block with the highest factor, in this case A with 0.7. All other blocks are associated with budgets initialized to zero. In every step a block of A is generated (since it has the highest factor) and its factor is added to the budgets of all other blocks. Then the budgets of B and C are decremented by its factors, respectively. If the budgets falls equal or below zero, then a block of B or C, respectively, is generated. In this example, after seven steps, the budgets of B and C start repeating, thus the pattern *AAAABAAABC* of blocks is repeated until 500 blocks have been generated.

IV. Case Study

We have implemented our proposed approach in Python. As a case study, we consider the SoCRocket VP [11][2]. SoCRocket is a power aware open-source VP written (primarily) in C++ (around 50k lines of code). We have extended SoCRocket to include a lightweight power layer that associates each component in the VP with a power FSM. Further we have added a *Power Interface Unit* (PIU) as described in the preliminary section which allows the FW to control the power transitions of each component in the VP.

The SoCRocket VP consists of various components including a LEON3-based CPU, an interrupt controller, a UART interface and a memory with corresponding controller, connected by an AMBA-based bus system. For this case study we have added a special processing unit (SPU) which allows to perform special operations independent of the CPU. The SPU is configured through memory mapped (MM) writes. The CPU can either actively wait for the result (spin) of the SPU or sleep until the SPU triggers an interrupt. Further, the SPU is a bus master by itself and thus can independently access the memory.

We consider a three dimensional cross power FSM combining the FSMs of the CPU, memory (including memory controller), and the SPU, respectively. Thus, our goal load interval vector (GLIV) is a cuboid, representing the duty cycles of these three components. Every component can be either in full power mode (FP) or one of four power safe modes: PS0, PS1, PS2 or PS3, with PS0 being the least and PS3 the most power saving mode. Thus, the complete cross power state space in this case study consists of 5*5*5 = 125 (interdependent) states.

In the following we first introduce the individual (instruction) blocks that we use and provide the pre-computed calibration information. Then we present the results of our experiments using four different PM strategies.

A. Block Definition and Calibration

Table II shows the individual block informations. The first row shows the load vector measured on the VP for the CPU, memory and SPU, when executing the block exclusively with all components of the VP being in full power mode (so the FW-based PM strategy is switched of for this measurement). The reason that we do calibration in full power mode only, is to avoid the state explosion of considering all combinations of power states for all different components in the system. The second row shows the average runtime in nanoseconds (NS) for executing only one block.

In total we have defined eleven different blocks. Two blocks that perform arithmetic operations based on addition, subtraction inside of loops. Four blocks that primarily perform various memory operations, including swapping and copying memory elements. A sleep block that will power down the system, hence reducing the duty cycles of the components. Four blocks to interact with the SPU: The CPU can sleep or spin and the SPU can either access the memory to perform the computation or not. When sleeping, the CPU will wakeup by an interrupt triggered from the SPU, and when spinning, the CPU will actively keep polling the SPU using memory mapped IO.

For illustration, Fig. 5 shows 1) an arithmetic block, 2) a memory block and 3) a block accessing the SPU. The first block performs addition and subtraction inside a loop. A *volatile int* argument is passed in from the main function (which is just a local variable initialized with a constant value) to avoid pre-computing the result by the compiler due to optimizations. The return value from the function is ignored from within the main

[2]See `www.systemc-verification.org` for our most recent VP-based approaches.

TABLE II: Individual block information obtained by running the corresponding block exclusively on the VP (all components in full power mode, i.e. no PM enabled in FW) and measuring the duty cycles (load) and average runtime.

	Arithmetic	Arithmetic2	Sleep	Memory	Memory2	Memory3	Memory4	CpuSleepNoMem	CpuSleepWithMem	CpuSpinNoMem	CpuSpinWithMem
Load (CPU, MEM, SPU)	(100, 1, 0)	(20, 81, 0)	(2, 1, 0)	(65, 44, 0)	(76, 34, 0)	(51, 54, 0)	(53, 53, 0)	(7, 8, 96)	(3, 86, 96)	(68, 8, 100)	(33, 83, 100)
Avg. Runtime in NS	465330	424360	1069000	291080	384950	133850	336990	1123760	1104760	1083230	1155440

```
1 #define REPEAT_10(x) do { x; x; x;
      x; x; x; x; x; x; } while (0);
2 #define REPEAT_100(x)
      REPEAT_10(REPEAT_10(x););
3
4
5 int ArithmeticBlock(volatile int x) {
6   int sum = 0;
7   int i;
8   for (i=1; i<2500; ++i) {
9     sum += i; sum -= x;
10  }
11  return sum;
12 }
13
```

```
14 void MemoryBlock() {
15   volatile int a[1024];
16   int k;
17   int x;
18
19   for (k=0; k<10; ++k) {
20     REPEAT_100(
21       x = a[k];
22       a[512+k] = x;
23       a[k] = a[512+k];
24     );
25   }
26 }
27
28
```

```
29 void CpuSpinWithMemBlock() {
30   volatile char in[1024];
31   volatile char out[1024];
32
33   *SPU_inputaddr  = (uint)&in[0];
34   *SPU_outputaddr = (uint)&out[0];
35   *SPU_operation  = 256;
36   *SPU_running    = 1;
37
38   // keep spinning until result is
        there
39   while (*SPU_running) {
40     ;
41   }
42 }
```

Fig. 5: Example instruction blocks with different type for illustration: an arithmetic, memory and SPU access block.

function, it ensures that the computation of the result value is not discarded. In general one has to be careful when writing C code due to compiler optimizations/re-structuring, which can make it more difficult to define suitable blocks. This could be circumvented by writing the block code in assembler directly. Also please note, that writing the blocks has to be only done once. The second block performs multiple memory swap operations. The *REPEAT* macro is used to eliminate some loop checking and update operations, hence putting more weight on the memory operation. Again, to avoid compiler optimizations, the array, that is operated on, is declared volatile. The third block employs the SPU to perform some special operation. The SPU is configured per memory mapped (MM) access to read from and write to a specific memory region. A MM write to the the *running* register will start the SPU operation. The CPU is actively waiting (spins) until the result is available (indicated by a zero in the *running* register of the SPU). The SPU is itself a bus master and will perform various memory operations to perform its computation.

In general the blocks should be defined in such a way, to have different load values which cover the cross load state space as thoroughly as possible. This allows to cover the remaining gaps in the cross power state space by mixing the blocks in different combinations. In this work we consider lines between blocks (i.e. their load vectors) for mixing, though this can be further extended to planes or tetrahedra, etc. if necessary. With 11 different blocks we have a total of $\binom{11}{2} = 55$ line combinations. A line describes the load of the three components that can be achieved (in our model, which is a prediction of the actual VP loads) by mixing its endpoints.

B. Experiments

We have evaluated our approach on four different duty cycle based PM strategies. The strategies are implemented in FW and executed periodically during an update cycle:

1. *on-demand* : This strategy will gradually power down the component, but immediately transition to the full power mode when work is available. Fig. 1 shows the corresponding power FSM.
2. *conservative* : Will gradually power down (as the *on-demand* strategy) and also gradually power up the component, visiting each power state one after another. Thus, it takes multiple update cycles to fully power up a component which has been in a (deep) power saving mode.
3. *balanced* : Gradually powers down from FP to PS1 and gradually powers up from PS3 to PS1. From the PS1 state immediately switches to FP or PS3 when work is available or the component is idling, respectively.
4. *combined* : Use a different strategy for every of the three considered components: *on-demand*, *conservative* and *balanced* for the memory, SPU and CPU component, respectively.

In the *on-demand*, *conservative* and *balanced* setting all three components use the same strategy. We run the experiments on a Linux machine with a 2,4 GHz Intel processor and 32 GB Ram. Table III shows the results. The right half shows results for the four above mentioned PM strategies. The table is separated by double lines into two parts and a header.

The upper part (not the header) shows the information of using a random test generator (*Random-only*). The number of blocks is constrained to be between 1000 and 2000 for each test. The blocks itself are currently randomly generated. This table part shows the execution time of all tests together in seconds, the power state cross coverage achieved by executing the tests, and the number of tests generated and executed.

The lower part shows results for our proposed approach. It performs three steps one after another. First it starts by bootstrapping the coverage with random testing. This is the same as the *Random-only* approach but we only use 100 test-cases for this bootstrapping. The reason is that adding additional random tests does not increase the coverage very much (as the *Random-only* row has demonstrated). Then our coverage-loop is executed working in *point-mode* and *line-mode* as has been explained in Section B. The test generator is using a suffix X of 600 blocks. For every step we report the execution time in seconds as well as the total coverage obtained after the step. The last row in this lower part of the table shows the total execution time of all steps of our approach together.

It can be observed that random testing does not perform well on this problem instance. For example increasing the number of tests from 100 (see the bootstrap phase of our approach) to 1000 does increase the achieved coverage only marginaly (e.g. from 44% to 53.6% and from 18.4% to 21.6%), even though a large part of the cross power state space is still uncovered, and at the same time the runtime grows (roughly) linearly by a factor of 10. This result demonstrates that an approach performing random test generation is not suitable for our use case.

In contrast it can be observed from the results that our coverage-driven approach works very well to close the remaining coverage gap. Close to 100% power state cross coverage is achieved for every considered PM

TABLE III: Experimental results for our approach

Technique to maximze cross coverage			Duty cycle based power management (PM) strategy implemented in firmware (FW)			
			on-demand	conservative	balanced	combined
Random-only		time in sec.	14144.81	13795.99	13172.07	13104.20
		coverage	67 / 125 (53.6%)	27 / 125 (21.6%)	76 / 125 (60.8%)	61 / 125 (48.8%)
		num. tests	1000	1000	1000	1000
Our Approach	1) random (bootstrap)	time in sec.	1450.97	1381.74	1364.74	1338.39
		coverage	55 / 125 (44.0%)	23 / 125 (18.4%)	53 / 125 (42.4%)	49 / 125 (39.2%)
		num. tests	100	100	100	100
	2) point-mode	time in sec.	1617.09	1790.34	1680.92	2210.17
		coverage	118 / 125 (94.0%)	111 / 125 (88.8%)	113 / 125 (90.4%)	123 / 125 (98.4%)
	3) line-mode	time in sec.	3173.76	2698.69	2634.66	1401.87
		coverage	124 / 125 (99.2%)	115 / 125 (92.0%)	121 / 125 (96.8%)	125 / 125 (100%)
	total time in sec.		6241.82	5870.78	5680.32	4950.43

strategy with reasonable runtime overhead. Already the point-mode strategy is sufficient to achieve very high coverage of around 92% on these examples. Applying the line-mode strategy afterwards does increase the coverage further to up to 100% with an average coverage of 97%. For some PM strategies the coverage is still (slightly) below 100%. In general the reason is that either: 1) a *stronger* mixing model, or 2) different blocks that cover the cross load state space more thoroughly are required. In this work, the reason for uncovered cross states has been the second case, in particular, that we were unable to define blocks with a very high CPU and memory load at the same time (due to synchronization between them). Careful analysis reveals that most of these uncovered cross states are in fact unreachable: 9 out of the 10 uncovered cross states for the *conservative* PM strategy and all 4 uncovered cross states for the *balanced* PM strategy are unreachable. Only one reachable cross state has been missed for the *on-demand* and also only one for the *conservative* PM strategy by our approach. Thus, our approach achieves optimal and near optimal results for the considered PM strategies by employing our mixing model and block definitions. This experimentally proves our approach to be very effective in maximizing power state cross coverage.

V. Discussion and Future Work

Cross coverage of power states is an effective but at the same time challenging metric to evaluate the quality of a test-set in validating power and timing constraints. Scalability can be an issue, because the cross FSM can grow exponentially. However, in general the number of power states per component (and thus single FSM) is rather small. Furthermore, using a subset of (important) components, possibly in different combinations of small cross FSMs, can already provide very useful coverage results. Our experimental evaluation demonstrated the applicability and efficacy of our coverage-driven approach in maximizing the power state cross coverage. Nonetheless, there are still possible directions for further improvements:

1. Currently individual block information (calibration) are obtained with the system running in full power mode to avoid the state explosion of calibrating the system with all possible power state combinations. However, the real system will switch power states during execution and thus deviations from the calibration result can be obsered. A viable solution might be to calibrate the system only for the (cross) states of the considered cross FSM. The cross FSM is typically based on a small number of components (three in our case-study) and has arguably the biggest influence on the runtime prediction (since the goal is to maximize coverage of the cross FSM).
2. Caching and potentially other side effects, due to block mixing, can lead to deviations of the (static) calibration result at runtime. Thus, it seems useful to integrate dynamic information, observed at runtime during test-case execution, into the test generation process.
3. Integrate more sophisticated block mixing and refinement procedures. Our *line-mode* approach can be extended to e.g. *plane-mode* or *tetrahedra-mode*, in combination with barycentric coordinates, to allow for a more flexible mixing of blocks. This becomes particularly useful for larger and higher-dimensional cross FSMs.
4. Investigate the use of formal verification techniques at the abstraction level of VPs, e.g. [16, 17], to automatically identify unreachable cross states and to cover cross states that proved to be very difficult to reach with simulation-based methods.

References

[1] IEEE Std. 1666, *IEEE Standard SystemC Language Reference Manual*, 2011.
[2] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018.
[3] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[4] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, C. Ykman-Couvreur, D. Quaglia, F. Ferrero, and R. Valencia, "The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration," *Microprocessors and Microsystems*, vol. 37, no. 8, Part C, pp. 966 – 980, 2013.
[5] S. Schürmans, D. Zhang, D. Auras, R. Leupers, G. Ascheid, X. Chen, and L. Wang, "Creation of ESL power models for communication architectures using automatic calibration," in *DAC*, pp. 1–6, May 2013.
[6] G. Onnebrink, R. Leupers, G. Ascheid, and S. Schürmans, "Black box ESL power estimation for loosely-timed TLM models," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 366–371, July 2016.
[7] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.
[8] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *SoC*, pp. 1–7, 2012.
[9] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, pp. 1–8, 2017.
[10] V. V. Singh and A. Kumar, "Cross coverage of power states," in *Design and Verification Conference and Exhibition (DVCon)*, 2016.
[11] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the European Space Agency's SoC development," pp. 1–7, 2014.
[12] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, pp. 286–291, 2003.
[13] C. Ioannides and K. I. Eder, "Coverage-directed test generation automated by machine learning – a review," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, pp. 7:1–7:21, Jan. 2012.
[14] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, "A technique for test coverage closure using goldmine," *TCAD*, pp. 790–803, 2012.
[15] P. Dasgupta, M. K. Srivas, and R. Mukherjee, "Formal hardware/software co-verification of embedded power controllers," *IEEE Trans. on CAD*, vol. 33, no. 12, pp. 2025–2029, 2014.
[16] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, 2018.
[17] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, pp. 52:1–52:8, 2016.