Co-simulation Framework for Streamlining Microprocessor Development on Standard ASIC Design Flow

Tomoyuki Nakabayashi†, Tomoyuki Sugiyama†, Takahiro Sasaki†, Eric Rotenberg‡, and Toshio Kondo†

†Graduate School of Engineering, Mie University
Tsu, Mie, 514-8508, Japan
Tel: +81-59-231-9780, Fax: +81-59-231-9781
‡Department of Electrical and Computer Engineering, North Carolina State University
Raleigh, North Carolina, 27695-7911, USA
Email: {tomoyuki, sugiyama, sasaki, kondo}@arch.info.mie-u.ac.jp, ericro@ncsu.edu

Abstract— In this paper, we present a practical processor co-simulation framework for not only RTL simulation but also gate/transistor level simulation, and even chip evaluation with an LSI tester. Our framework includes an off-chip system call emulation mechanism, which handles system calls to evaluate and verify the processor design with general benchmark programs without pseudo-circuits in the processor design. Therefore, our framework can be consistently used from RTL design to chip fabrication. We also propose a checkpoint mechanism that resumes a program from a pre-created checkpoint. This mechanism is not affected by the non-deterministic problem on a multi-core processor. Moreover, we propose a cache warming mechanism when resuming from a checkpoint.

I. Introduction

As multi-core architecture has become commonly used to improve processor performance, designing a state-of-the-art multi-core chip in a short time has become essential for processor research. A development environment that contains useful mechanisms and can be used throughout the entire processor research provides efficient infrastructure to researchers. We classify the steps of fabricating a novel processor chip into five phases, 1) design space exploration using a simulator, 2) register transfer level (RTL), 3) gate level, 4) transistor level, and 5) fabricated chip. There are two challenges to streamline the processor development through the entire standard ASIC design flows.

1. *Emulating system calls in RTL through fabricated chip:* When researchers prototype a processor from RTL, to gate and transistor level, to ASIC, it is often desirable to focus on user level code because they are interested in the core part of the processor and not all of the system level support. They may be in this situation because they designed the RTL from scratch or because they are using open source toolsets (e.g., FabScalar) which provide a level of sophistication in the microarchitecture but do not currently feature system level support. As a matter of convenience, and a matter of research productivity, it is good to dispense with the issue of explicitly supporting system calls in the processor design. While emulation is often used in simulators written in a high-level language, it is unwieldy to carry that over to RTL/gate/transistor simulations and not at all possible to emulate in the same way for a fabricated chip.
2. *Reducing turnaround time through sampled execution*: Except for the fabricated chip phase, all other simulation-based phases in particular gate/transistor level cannot simulate the entire workloads in a reasonable timeframe. Therefore, we need a checkpoint mechanism to resume a simulation from an arbitrary region of interest (ROI). Moreover, checkpoints are useful when hardware bugs are detected in the fabricated chip. A checkpoint allows for bypassing the bug (if it is infrequent) to get to another ROI. Therefore, it is also useful for validation in the fabricated chip phase.

In this paper, we propose a co-simulation framework to address these challenges. This paper makes the following contributions:

1. Our framework includes an off-chip system call emulation mechanism that handles system calls using general load and store instructions. This enables the processor design to involve the execution of a program using system calls without booting an OS on the target processor for evaluation and verification. The off-chip system call emulation mechanism enables a prototype processor that cannot handle system calls to execute a program using system calls. Therefore, researchers can improve research productivity.
2. Our framework also contains a checkpoint mechanism to reduce turnaround time for evaluation and verification. The checkpoint mechanism restores not only essential state (register file, program counter, inflight file/network operations, and memory state of the process) but also optional state (warming up the caches).
    - *Essential state restoration:* Our checkpoint mechanism resumes a program from an ROI even on a multi-core processor chip. With this mechanism, researchers can shorten turnaround time and obtain the result in the ROI.

978-1-4799-2816-3/14/$31.00 ©2014 IEEE

- *Optional state restoration:* The checkpoint mechanism also contains a start-up routine to warm up the cache with the cache replacement algorithm. The warm up mechanism reduces the time to achieve a peak performance and also improves simulation accuracy by diminishing the effect of cold started cache.

Our co-simulation framework can be consistently used in RTL, gate and transistor level simulations, and in fabricated chip evaluation because all the above mechanisms are implemented without pseudo-circuits such as direct programming interface-C (DPI-C) in the processor design. In addition, our framework does not depend on the microarchitecture of a processor. We introduced our framework into two processor design projects: a simple single pipeline processor and a complex out-of-order processor.

II. Related work

A. Processor simulators

Many processor simulators [1, 2] and system simulators [3, 4, 5] written in a high-level language are used for processor research. Researchers take advantage of such simulators in the early stage of research in accordance with their intended use. Since our main focus is from RTL to fabrication, in which researchers evaluate the precise hardware cost, energy efficiency, and circuit delay for their proposed approach, we describe three focused mechanisms: 1) system call emulation to simplify processor architecture, 2) checkpoint mechanisms to reduce simulation time, and 3) cache warming mechanism to achieve a highly accurate evaluation. These mechanisms, however, are used only in each simulator. Our goal is to use these mechanisms in all phases of standard ASIC design flows.

B. Synthesizable processors

Some open synthesizable processors can be used from RTL implementation to chip fabrication [5, 6, 7, 8]. Since FabScalar and OpenSPARC have a co-simulation environment, we describe these two processors in more detail.

FabScalar automatically generates synthesizable RTL designs of differently designed superscalar cores. FabScalar contains an instruction set simulator, called functional simulator, to verify RTL by concurrently running the same instructions in RTL design and the functional simulator, and cross-checking the architectural state instruction-by-instruction. The functional simulator is also used for emulating a system call, so RTL design can handle a system call as a one-cycle-instruction. Also FabScalar provides fast-skip and checkpoint mechanisms to avoid long simulation time and re-simulating up to a checkpoint. However, FabScalar currently has drawbacks in system call emulation (described in Section IV) and the checkpoint mechanism (described in Section V). Our aim was to improve the two mechanisms based on FabScalar.

OpenSPARC is the open-source version of UltraSPARC T1 and T2 processors. Currently, RTL design, simulation tools, and verification package are all available. OpenSPARC provides a complete RTL design to boot a full OS and useful tools for simulation and verification including checkpoint and cache warming. However, there is the level of abstraction gap as the next step from a processor simulator, and this gap makes it difficult to advance the research phase beyond simulator-based exploration. By contrast, our off-chip system call emulation mechanism enables a designer to evaluate a prototype processor omitting touchy hardware for an OS with general benchmark programs. In addition, cache warming of OpenSPARC is implemented by programming language interface (PLI) in verilog, this limits the use to only in RTL simulation. Our cache warming mechanism is unique in that it is consistently used from RTL to fabricated chip.

III. Co-simulation framework

A. Co-simulation overview

This subsection gives an overview of our co-simulation framework. The framework consists of a *functional simulator* written in a high-level language and *processor design*. Note that *processor design* refers to all designs of RTL, gate, transistor level, and fabricated chip. Fig. 1 shows how the functional simulator is used in our framework. The functional simulator assists in the verification and evaluation of processor design. The cross-checking architectural state guarantees instruction set level behavior of processor design, and fast-skip and checkpoint mechanisms reduce turnaround time. In addition, the functional simulator emulates system calls by calling the host OS according to a request from processor design.

B. Challenges

Three challenges in the co-simulation framework are described below.

System call emulation: System call emulation is significantly beneficial in that a designer can run a general program without booting an OS on the target processor. Our system call emulation mechanism described in Section IV can be used for every research phase. Our framework exploits general load and store instructions to communicate with the emulator; therefore, no special mechanism is necessary in the processor design.

Checkpoint mechanism: The checkpoint mechanism is used not only to reduce turnaround time but also to evaluate only an ROI. Checkpoint creation for a multiprocessor should consider the non-deterministic problem. In Section V we propose a checkpoint mechanism that solves this problem.

Cache warming: Our checkpoint mechanism contains cache warming mechanism as a part of optional checkpoints. Restoring a checkpoint is exposed to a large performance gap with a peak performance because the simulation is resumed with a cold started cache. In addition, in the gate and transistor level phases, it takes a long time to achieve the peak. Our cache warming mechanism described in Section VI warms up the cache in the shortest time and improves evaluation accuracy.

We introduced our framework into two processor design projects: an embedded processor [9] and FabScalar.

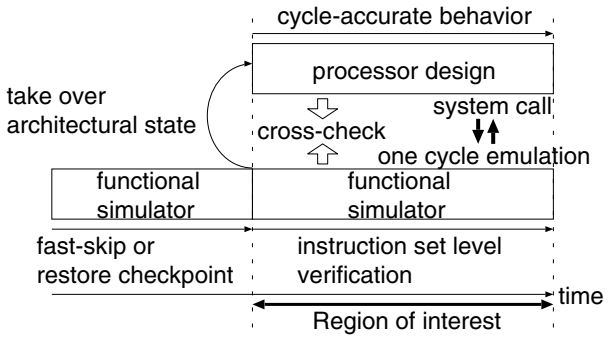

Fig. 1. Co-simulation framework.

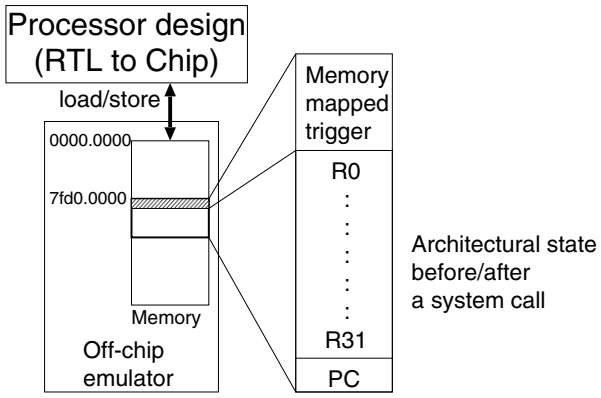

Fig. 2. Off-chip system call emulation mechanism.

IV. System call emulation

In general benchmark programs such as SPEC, a processor must handle system calls (services from an OS kernel) to handle the file system, network, memory, process, thread, and security. For this reason, to evaluate a processor design with general benchmarks, the processor either boots an OS or uses an alternative stand-alone C library. There are two requirements. One is that researchers directly execute benchmarks on a full implemented processor to evaluate and verify in a short time. Since booting an OS takes a large amount of time, especially in gate and transistor level, it is difficult to evaluate or verify a processor design on a full system. Moreover, although using an LSI tester has an advantage of directly evaluating or testing a fabricated chip with input vectors, such LSI testers limit the execution cycle up to insufficient cycles for running a target application. The other requirement is that to evaluate a microarchitectural approach, researchers often require only primal instructions such as arithmetic, logical, branch, and memory access instructions, and tends to omit subsidiary hardware for an OS such as memory management unit and internal processor registers. Implementing such hardware requires researchers to have a deep understanding of such hardware.

Because of these two requirements, executing general programs without an OS is valuable. Newlib is a C library intended for use on embedded systems [10]. A processor can execute programs without an OS with the addition of a few low-level routines. However, another binary file using Newlib is needed. In addition, Newlib requires emulation of peripheral systems and does not support multiple processes and cores.

Emulating a system call as an instruction solves the above problems. When a system call occurs, the functional simulator detects the call and emulates it by calling the host OS. Later, the processor design continues execution after reflecting the result of the system call.

To emulate a system call, the processor design somehow notifies the emulator of the occurrence of the system call. Furthermore, the processor design must take over the system call result. In the RTL phase, this is not difficult because a test module can look into the submodule, which asserts a relative signal, and overwrite the architectural state. FabScalar currently uses this method; however, it is used only in the RTL phase and prevents regions including system calls from being evaluated on FPGA [11]. Therefore, our framework is necessary for emulating system calls beyond the RTL phase.

A. Implementation of off-chip system call emulator

The concept of our system call emulation mechanism is juggling a system call as consecutive stores and loads. We explain our emulation mechanism using Figs. 2 and 3. Fig. 2 shows memory mapping and how to trigger/reflect a system call emulation. We allocate a memory space (e.g., from address *7fd00000*) to interact with the off-chip emulator. First, when a system call occurs, the processor jumps to the system call trap routine like a real product. We use the routine shown in Fig. 3 instead of a true routine if an user wants to emulate system calls. Second, the processor design involves storing the architectural state, i.e., register file, to the prescribed space because the emulator requires the register file values to emulate the required system call. Next, the emulator emulates the system call when a store is executed into the probing address (*7fd00000*). Finally, the emulator writes the values updated by the system call into the same memory space to which the processor stored the register values, then the processor loads the modified values using load instructions. Note that if the processor includes a cache, the stores and loads should be non-cacheable memory access instructions. This emulation mechanism does not require any dedicated hardware in the processor design; therefore, the processor maintains a pure design. Since the processor interacts with the emulator using load and store instructions in our framework, the emulation mechanism can be consistently used from RTL to fabrication. This enables the processor design to execute a general program without booting an OS.

The off-chip system call emulation mechanism is of course used for evaluating and verifying a complete processor. Also, a prototype processor design which does not support system calls can be evaluated with general benchmarks. This aspect improves research productivity to evaluate a microarchitectural approach.

V. Checkpoint mechanism

A checkpoint mechanism saves the state of a simulation in an ROI and later continues the simulation from the ROI. A checkpoint mechanism goes through two phases: checkpoint

```
bfc003d0 <__trap_syscall>:
        /* Save architectural state */
        sw $1,  0x104(k0)

        sw $31, 0x17c(k0)
        /* Trigger for system call */
        sw 0x01, 0x0000(k0)
        /* Restore the result */
        lw $1,  0x104(k0)

        lw $31, 0x17c(k0)
```

Fig. 3. System call trigger routine

```
bfc00000 <__reset_handler>:
        lui k0,  0x7fd0
        lw  $1,  0x104(k0)

        lw  $31, 0x17c(k0)
        /* load program counter */
        lw  k1,  0x278(k0)
        jr  k1
```

Fig. 4. Reset routine

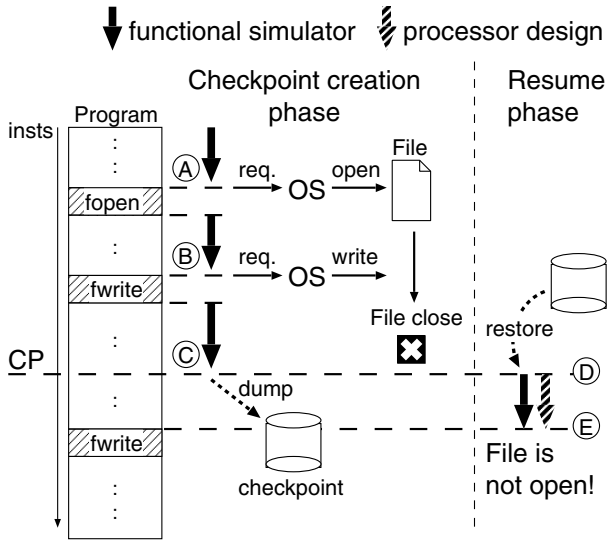

Fig. 5. Problem with checkpoint mechanism.

creation phase with only the functional simulator, and resume phase with the processor design. We can repeat the resume phase during verification and evaluation of a processor to reduce turnaround time.

In our co-simulation framework, we resume a program using a similar routine as the system call emulation for use in every design phase to restore the architectural state. We use the reset routine shown in Fig. 4. After the processor is reset, the program counter is initialized to *bfc00000*, which is the general start address of the reset routine. In the routine, the processor loads register file values and the program counter written into the prescribed memory space. The program counter indicates the starting point of a checkpoint.

However, if the co-simulator naively resumes a benchmark program from a checkpoint, a system call (file- and network-related) cannot be correctly executed. In the following explanation, we use a sequence of file system operations as an example to simplify the problem. The co-simulator leaves file input/output (I/O) to the OS running on a host computer. Fig. 5 shows the issue of resuming simulation from a checkpoint. When a file is opened, the off-chip system call emulation mechanism calls the OS to handle the file opening (Fig. 5.A). Once the file is opened, the co-simulator treats I/O operation to the file in the same way (Fig. 5.B). Once the simulation reaches at the start point of an ROI, the co-simulator creates the checkpoint, then the file is closed because the co-simulator quits (Fig. 5.C). For this reason, when the co-simulator resumes the simulation from the checkpoint (Fig. 5.D) and a file I/O occurs (Fig. 5.E), the co-simulator cannot handle the file I/O because the file is not open.

To solve this problem, FabScalar dumps the state not only at a checkpoint but also at file I/Os in an ROI as shown in Fig. 6. In the checkpoint creation phase, FabScalar executes a program beyond a checkpoint to dump the state after file I/Os in the ROI (Fig. 6.A). To resume a program, FabScalar restores the state at the checkpoint (Fig. 6.B). When a file I/O occurs during the resumed simulation, the dumped state after the file I/O is restored (Fig. 6.C); therefore, FabScalar reproduces the state after the file I/O. With this method, FabScalar provides a checkpoint mechanism. However, FabScalar can execute only file I/Os that were pre-executed in the checkpoint creation phase. In addition, this mechanism cannot be applied to a multiprocessor environment because we cannot create the preceding checkpoint. Because the execution order is non-deterministic in a multiprocessor, the order of system calls is also non-deterministic.

There are a few solutions to the problems. M5 uses the solution of dumping all necessary information into a checkpoint file, e.g., the offset of the file descriptor manipulated in the simulation. By contrast, we adopt another solution because M5's solution requires the dumping all inflight operations such as file system and network. In our solution, the simulator executes only related system calls to resume a program with inflight operations. Fig. 7 shows our solution. Our co-simulator dumps the difference in state between each file I/O up to a checkpoint in the checkpoint creation phase (Fig. 7.A). It skips the instructions between each file I/O using the dump file and executes only file I/Os (Fig. 7.B). After it reaches the checkpoint, it continues execution including file operations (Fig. 7.C). As a result, our co-simulator skips to a checkpoint at high speed without any restriction.

Although our solution has the advantage of handling all inflight operations in the same way, restoration speed depends on the number of system calls up to a checkpoint. To demonstrate that our solution is practical, we evaluated the restoring of speed using SPEC2000 INT benchmarks. We created a checkpoint in SimPoint [12] and resumed the checkpoint for each benchmark program. Table I lists the evaluation results. The upper half compares the time to forward each benchmark to the SimPoint. We evaluated all benchmarks on Intel Core i7-2600 CPU @ 3.40 GHz with 4 GB memory. We used a 4-width

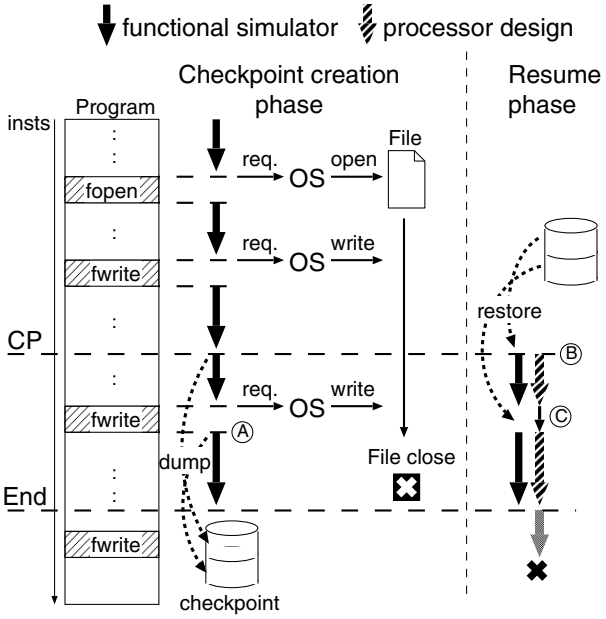

Fig. 6. Checkpoint mechanism of FabScalar.

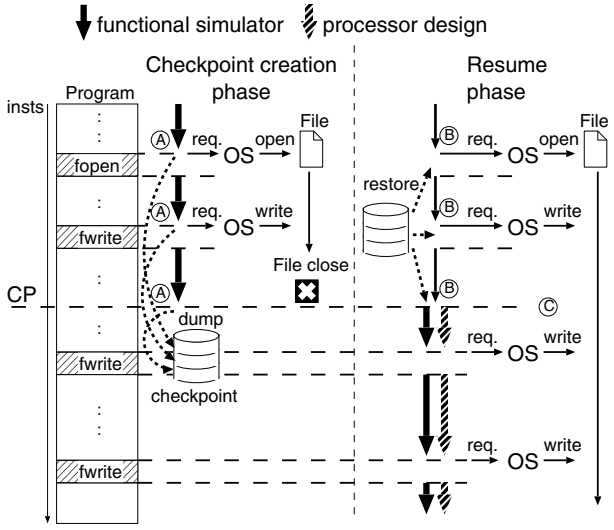

Fig. 7. Proposed checkpoint mechanism.

TABLE I
DEMONSTRATION OF CHECKPOINT MECHANISM.

	gzip	mcf	bzip	parser	twolf
Gate-level[a] (day)	13,757	6,389	11,297	13,260	12,319
RTL design[a] (day)	844	326	715	680	645
fast-skip (min.)	244	103	206	210	212
checkpoint (sec.)	0.50	0.68	0.77	3.84	0.53
skipped insts (100 million)	1,189	553	977	1,146	1,066
checkpoint file size (MB)	832	326	384	1780	119
system calls	65	116	101	1,027	133

[a]Estimated by million instructions per second (MIPS) value

fetch superscalar processor design as the RTL design and synthesized the RTL design for the gate-level estimation. We used Cadence NC-Verilog, version 09.20-s038, for simulation and Synopsys Design Compiler, version H-2013.03-SP2, for synthesis. We note that checkpoint restoration succeeded in both RTL and gate-level simulations. The lower half of the table summarizes the number of skipped instructions, the file size of the checkpoint, and the number of system calls up to the SimPoint. The results show that restoring a checkpoint took a few seconds in the worst case and the file size of the checkpoint was not so large.

VI. CACHE WARMING MECHANISM

A cache system has a large impact on processor performance. When we resume a benchmark program from a checkpoint, a cold started cache incurs a performance gap with peak performance, as shown in Fig. 8. Therefore, the evaluation accuracy is degraded because of the performance gap. Furthermore, it takes a long simulation time to warm up a cache system to analyze the peak performance/energy. OpenSPARC has a cache warming mechanism using PLI in verilog HDL. This implementation limits the use to only in the RTL phase. By contrast, our cache warming mechanism can be used in all design phases. It is particularly effective in shortening the test vector for an LSI tester. In addition, our cache warming mechanism defines a certain time when the cache system is warmed up, this feature enables a designer to evaluate only a specified period after the processor achieves the peak in simulation, as shown in Fig. 9.

Fig. 9 shows our cache warming mechanism. Our co-simulator also has a cache simulator written in C language. When the co-simulator creates a checkpoint, the cache system dumps the cache warming routine (binary file, actually), as shown in Fig. 9. Lines that are accessed with the same index are dumped in order of the least recently used (LRU) value to restore the cache contents including the cache replacement algorithm. We depict that a lower LRU value has a higher priority for replacement, i.e., an entry whose LRU value is *0* will be replaced. The dumped routine is linked when the co-simulator starts restoration, and it is called in the reset routine before restoring the architectural state. This mechanism restores the cache contents by using a software level approach in the shortest time.

We briefly estimated the impact of our cache warming mechanism on an L1 data cache (total size: 16 KB and line size: 16 bytes). Even in the worst case (cache warming on blocking cache), our cache warming mechanism reduced 90% of the simulation time to stabilize performance compared with a simulation using cold started cache. As a result, we can cut 15 minutes in gate-level simulation; therefore, we will be enable to skip tens of hours for a peak performance evaluation in transistor-level simulation. Our cache warming mechanism reduces one more order of magnitude when we use a non-blocking cache for cache warming.

Currently, optional checkpoints are only for data cache warming. Expanding optional checkpoints such as instruction

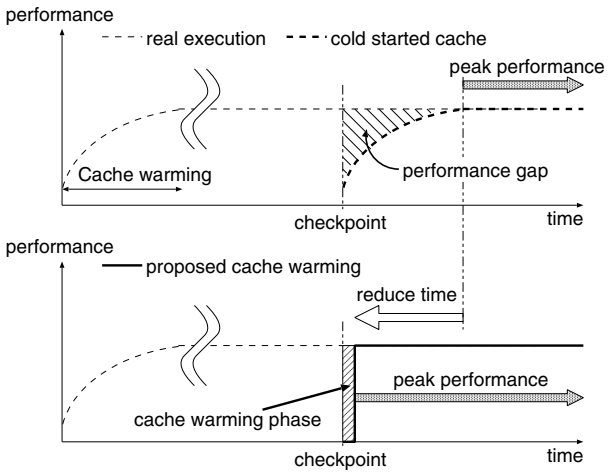

Fig. 8. Impact on performance using cache warming.

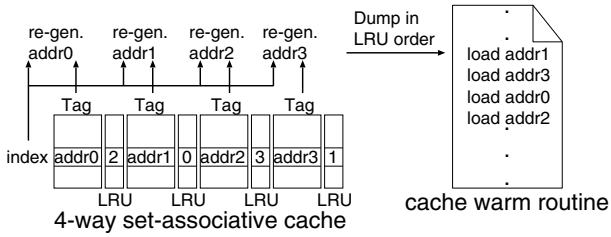

Fig. 9. Generating cache warming routine.

cache and branch predictors is left for future work.

VII. Summary and Conclusions

We proposed a practical processor co-simulation framework that provides system call emulation, checkpoint, and cache warming mechanisms through the RTL, gate level, transistor level, and fabricated chip phases. All the mechanisms were effective in two processor design projects.

For future work, we intend to demonstrate that our framework can be used for a fabricated chip we are currently designing. Also, the entire co-simulation environment will be available in the near future.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 24700047. This work is supported by the VDEC of the University of Tokyo in collaboration with Synopsys, Inc., Cadence Design Systems, Inc., and Rohm Corporation.

References

[1] D. Burger and T. M. Austin, "The Simplescalar tool set, version 2.0. technical report" *CS-TR-1997-1342, University of Wisconsin-Madison*, 1997.

[2] R. Shioya, M. Goshima, and S. Sasaki, "The design and implementation of processor simulator Onikiri2", *the Annual Symposium on Advanced Computing Systems and Infrastructures, poster*, 2009.

[3] F. bellard. QEMU: open source processor emulator. http://bellard.org/qemu/.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: modeling networked systems", IEEE Micro, 26:52-60, 2006.

[5] N. Fujieda, T. Miyoshi and K. Kise, "SimMips: A MIPS system simulator", *Workshop on Computer Architecture Education held in conjunction with MICRO-42*, pp. 32-39, December 2009.

[6] XUM Version 2.0: the eXtensible Utah Multicore Project, September, 2012. http://www.cs.utah.edu/formal_verification/XUM/.

[7] OpenSPARC: http://www.oracle.com/technetwork/systems/opensparc/index.html.

[8] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template", *Proceedings of the 38th IEEE/ACM International Symposium on Computer Architecture (ISCA-38)*, pp. 11-22, June 2011.

[9] T. Sugiyama, T. Sasaki, T. Nakabayashi, and T. Kondo, "Development of C++/RTL co-simulation environment for accelerating VLSI design of an embedded processor", *Proceedings of the 28th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2013)*, pp. 281-284, July, 2013.

[10] Newlib http://sourceware.org/newlib/.

[11] B. H. Dwiel, N. K. Choudhary, and E. Rotenberg. "FPGA Modeling of Diverse Superscalar Processors", *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*, pp. 188-199, April 2012.

[12] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior", *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 45-57.