Cycle-Count-Accurate Processor Modeling for Fast and Accurate System-Level Simulation

Chen-Kang Lo, Li-Chun Chen, Meng-Huan Wu, and Ren-Song Tsay
Department of Computer Science
National Tsing-Hua University, Hsinchu, Taiwan
{cklo, lcchen, mhwu, rstsay}@cs.nthu.edu.tw

***Abstract*—Ideally, system-level simulation should provide a high simulation speed with sufficient timing details for both functional verification and performance evaluation. However, existing cycle-accurate (CA) and cycle-approximate (CX) processor models either incur low simulation speeds due to excessive timing details or low accuracy due to simplified timing models. To achieve high simulation speeds while maintaining timing accuracy of the system simulation, we propose a first cycle-count-accurate (CCA) processor modeling approach which pre-abstracts internal pipeline and cache into models with accurate cycle count information and guarantees accurate timing and functional behaviors on processor interface. The experimental results show that the CCA model performs 50 times faster than the corresponding CA model while providing the same execution cycle count information as the target RTL model.**

I. INTRODUCTION

As both system-on-a-chip (SoC) design complexity and time-to-market pressure increase relentlessly, system-level simulation emerges as a crucial design approach for non-recurring engineering (NRE) cost saving and design cycle reduction. With system components, such as processors and busses, modeled at a proper abstraction level, system simulation enables early architecture performance analysis and functionality verification before real hardware implementation.

To construct a proper system platform for simulation, models for system components of various abstraction levels are proposed for simulation accuracy and performance trade-off. For example, cycle-accurate (CA) models are proposed to eliminate detailed pins and wires to improve simulation performance while preserving cycle timing accuracy. CA models are suitable for tasks, such as micro-architecture verification. The verification of correctness involves detailed states, such as values of register contents at every cycle. In practice, the simulation speeds of CA models are slow because of the enormous number of simulated states and are not satisfactory for system-level simulation.

To further increase simulation performance while sacrificing timing accuracy, cycle-approximate (CX) models apply simple fixed, approximated delays to represent timing behaviors. CX models achieve significant simulation performance speedup and are useful for architecture performance estimation at early design stages. Nevertheless, the approximated timing is inadequate for system simulation such as HW/SW co-simulation or multi-processor simulation. In such simulations, maintaining correct temporal execution order of concurrently executed system components relies on precise timing information [5, 10]. Without precise timing information, both performance evaluation and functionality verification cannot be accurate. For example, timing-related bugs, such as FIFO overflow in HW/SW interface, would not be faithfully reproduced as reported in [9].

A new modeling approach, i.e., cycle-count-accurate (CCA) approach, has received great attention lately, offering considerable simulation performance speedup compared to CA models (e.g., one order faster) by eliminating unnecessary timing details while keeping only needed system timing information [11, 12]. Compared to CX, CCA technique preserves accurate *cycle count* information of execution behaviors, and the preserved accuracy is adequate for system simulation.

Ideally, a CCA system simulation platform, with all components in CCA models, can perform quickly and accurately. Nevertheless, if certain components have no CCA models, then designers must mix CA or CX models with CCA models for system simulation. Consequently, either the simulation performance will be slowed down due to the slow CA models or the results will be inaccurate due to the inaccurate CX models. Unfortunately, there are no known *CCA processor models* to the best of our knowledge, although CA and CX techniques have been widely applied in processor modeling [2-4, 6-8].

In order to complete a fast and accurate CCA system simulation platform, we propose a CCA processor modeling technique in this paper. The idea is essentially based on the observation that, if the timing and functional behaviors of every access (such as bus access) on a component interface are correct, the effects from the component to the simulated system behaviors will remain correct. In other words, unnecessary internal component details can be eliminated to achieve better simulation performance while maintaining accurate system behaviors, as long as the interface behaviors are correct.

Essentially, the proposed CCA processor model preserves accurate cycle count information between any two

978-3-9810801-7-9/DATE11/©2011 EDAA

consecutive external interface accesses through pre-abstracted processor pipeline and cache timing information using static analysis.

Based on the proposed approach, a CCA model for a real OpenRISC 1200 processor is generated, and the experimental results show that the model performs 50 times faster than the corresponding CA model while providing the same execution cycle count information as the target RTL model.

The rest of this paper is organized as follows. After reviewing related work in section 2, the concept of CCA processor model is introduced in section 3. Then, sections 4 and 5 present the proposed processor modeling methodology based on the CCA concept. Finally, section 6 shows a case study with experimental results, and section 7 gives a brief conclusion.

II. Related Work

To provide accurate timing information for performance evaluation and functionality verification, the CA processor modeling technique is widely adopted [6-8]. For instance, Guerra et al. [6] propose integration of CA processor models and hardware models for HW/SW co-verification.

Although it is straightforward to use CA processor models for accurate simulation, excessive modeling details slow down simulation performance significantly. In contrast, the proposed CCA processor models maintain just enough internal details to achieve considerable simulation speedup while preserving accurate system behaviors.

For fast SW performance evaluation, CX processor models [2-4] adopt statistical delays to represent the timings of program segments, such as functions or basic blocks. Usually, given a target processor architecture, a statistically estimated fixed delay is annotated to each program segment. During simulation, the estimated execution time of the program is calculated by summing up the annotated delays along the execution path.

As a result, CX processor models achieve very high simulation speeds and serve the needs of software performance estimation at early design stages well. Nevertheless, the approximated timing of CX models can result in inaccurate system simulation results. That is why the CCA processor models are proposed to ensure correct simulation results while maintaining simulation efficiency.

In general, the CCA modeling idea is to speed up simulation by leveraging the limited *observability* of a system component and eliminating unnecessary internal timing details that do not affect the accuracy of the overall system simulation. The idea has been successfully applied to modeling hardware components such as buses and memories [11, 12] but not on processors yet.

For example, Sudeep et al. [11] describe a method for constructing an AMBA bus model that is cycle count accurate at the transaction boundaries (CCATB), and Lo et al. [12] apply the CCA concept to memory read/write transaction modeling. These two approaches maintain accurate timing at the beginning and the end of each bus/memory transaction and eliminate unnecessary intra-transaction states without compromise of system timing accuracy.

In contrast, the contribution of this paper is to extend the CCA concept to processor modeling. The proposed CCA processor model takes the time points of issuing external accesses as the (processor) transaction boundaries and correctly maintains these time points for accurate interface access executions. Accordingly, the timing correctness of the overall system is guaranteed through the accurate interface access behaviors.

Next, we will go into details and explain how a CCA processor model can be actually constructed.

III. Cycle Count Accurate Processor Modeling

The key idea of the CCA modeling technique is to leverage limited observability of component internal states and speed up simulation by eliminating unnecessary internal modeling details without affecting overall system simulation accuracy. In the following, we first discuss the observability property of processor models and then propose a CCA processor model.

For a processor component, only the behaviors on its interface are directly observable to the system (or specifically, to the rest of the system). In other words, a system cannot directly observe and interact with a processor except through the interface.

For the purposes of illustration, we examine a typical processor model in Fig. 1(a). We demonstrate a case where an instruction inside the pipeline requests *writing data* to the HW component. To accomplish the request, the data transferred has to pass through the cache and triggers a bus transfer action on the bus interface (BIF). A sample timing diagram of the bus transfer is shown in Fig. 1(b) for reference. In the transfer process, none of the processor internal behaviors, such as those of the pipeline and cache, can directly affect that of the HW component except through the

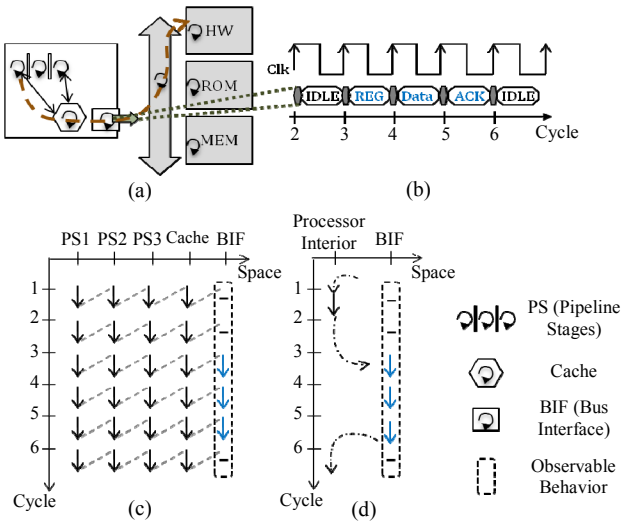

Figure 1: (a) An exemplified system; (b) A sample timing diagram on the bus interface; (c) The execution behavior of a CA processor model; (d) The execution behavior of a more abstract processor model.

bus access on the interface. In other words, the interface behavior (i.e., the bus access with the data transferred in this example) determines the effects from a component to the system.

The fact of limited observability implies that, if two processor models have the same interface behaviors, they have equivalent effects on the system. Naturally, the more efficient model should be used for system simulation. This motivates us to find a more abstract and efficient processor model than CA models.

For illustration, Fig. 1(c) and 1(d) respectively show a CA and a more abstract processor models. Although they have different internal execution details, both models display the same bus access behaviors as shown in Fig. 1(b). Each column shows the behavior of a concurrent process, such as a pipeline stage (PS), and each arrow denotes a state evaluation of a process at the numbered clock cycle time. The CA model in Fig. 1(c) captures all the concurrent behaviors of the processor by updating every process state at every clock cycle; in contrast, a more abstract model in Fig. 1(d) gives same effects to the system by providing equivalent bus access behaviors.

This leads us to create a CCA processor model that provides exact timing in terms of *cycle count* on every external interface access point with simplified internal models. By eliminating unnecessary details using CCA processor models, the whole system simulation can both preserve perfect timing accuracy and gain significant simulation performance improvement.

To generate such a CCA processor model, we observe that all external accesses are initiated from the processor pipeline, and then pass through the caches to the processor interface. Hence, the proposed CCA processor model is constructed with an abstract *pipeline subsystem model (PSM)*, which issues access events at correct time points, and a *cache subsystem model (CSM)*, which simulates the caches with the access events and triggers external interface accesses accurately.

The details of PSM and CSM are elaborated in sections 4 and 5, respectively.

IV. Pipeline Subsystem Modeling (PSM)

To eliminate unnecessary simulation details of the PSM, we statically analyze all possible *pipeline execution behaviors* (PEBs) of each basic block of a given program. Then at simulation, the actual time points of issuing access events (to the CSM) are calculated based on the pre-analyzed PEBs. The static timing analysis is discussed in section IV.A while dynamic effects such as cache miss penalty are in section IV.B.

A. Static Timing Analysis

1) *Pipeline Execution Behavior (PEB) Analysis:* Ideally, with an abstract pipeline model capturing target pipeline architecture, the pipeline execution of any given fixed sequence of instructions can be statically determined [16]. Nevertheless, a complete program cannot be statically analyzed because it contains branches determinable only at runtime.

Hence, the static analysis pre-analyzes each basic block of the program since it contains no branches. For example, a control flow graph (CFG) in Fig. 2(b) is first constructed after analyzing a program in Fig. 2(a). Then, if we assume a target processor with a 4-stage pipeline, the PEB of basic block C can be analyzed as shown in Fig. 2(c). The scheduling result of pipeline executions is recorded on a table where its columns represent the pipeline stages and its rows represent cycle times. In this example, a *Bubble* (i.e., NOP) is inserted in the final pipeline execution to resolve the data hazard between instruction 7 and 8.

Actually, a basic block may have several possible PEBs because its execution could be affected by the executions of its precedent basic blocks. For example, further assuming the processor is equipped with a branch predictor, there would be two possible PEBs for basic block C, as the one previously analyzed in Fig. 2(c) and a new one in Fig. 2(d). Fig. 2(c) is the case when the branch prediction fails and the pipeline is flushed and hence basic block C is executed alone. On the other hand, if the branch prediction succeeds, the basic block C is executed immediately following the basic block A, as shown in Fig. 2(d). The resolution of the data hazard introduced by instructions 4 and 5 across basic blocks induces an additional delay and produces a different PEB for basic block C.

For efficient PSM simulation, all possible PEBs of every basic block are pre-analyzed. Given a program's CFG, the static analysis finds all strings of precedent blocks (or upward combinations of consecutive precedent blocks) that may induce different PEBs. In fact, the number of PEBs is bounded by the target pipeline length. This is due to the fact that if a precedent block is too far away from the currently analyzed block so that the instructions of the two blocks cannot be executed simultaneously in the pipeline, then it will not contribute to creating a new PEB.

As an example, we assume that basic block D in Fig. 2(b)

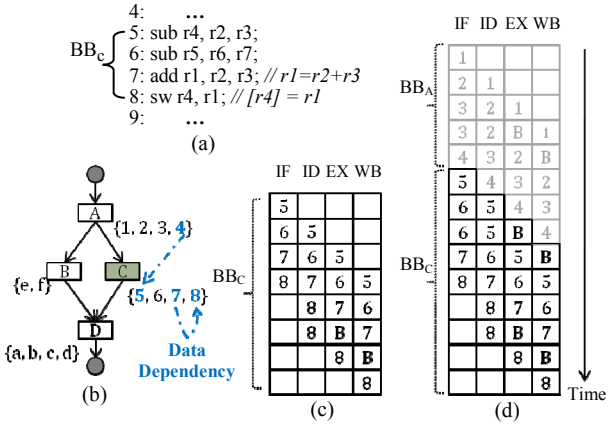

Figure 2: (a) A program segment; (b) A CFG of the program; (c) The PEB of basic block C alone; (d) The PEB of basic block C following basic block A.

is a block being analyzed. Tracing back the strings of precedent blocks through the left path of block D, we find that the combination {D, B, A} may induce a differnet PEB from the one induced by {D, B}, because block B only has two instructions, less than the pipeline length (i.e., 4), and block D could be executed with blocks B and A in the pipeline at the same time. Nevertheless, the combination {D, C, A} must produce the same PEB as {D, C}, since block C has four instructions, equal to or more than the pipeline length, and hence block A is too far from D through the right path to have both executed simultaneously.

In summary, for each basic block to find all possible PEBs, the static analysis traverses backwardly (e.g., using depth-first search) to find precedent block strings and compute the corresponding PEBs. It stops traversing deeper when the total number of the instructions on the string found is equal to or greater than the pipeline length.

2) *Access Timing Analysis for each PEB:* For efficient PSM simulation, we further statically analyze the (cache/IO) access timing behavior of each PEB by identifying both instruction and data access events at their corresponding execution time points.

For *instruction access events*, we check each instruction at the IF stage in PEB, because it basically indicates an instruction cache (I-cache) access occurred at that time point. However, only instruction accesses which may potentially cause cache misses should be identified as access events for simulation, since only they could cause external accesses and affect interface behaviors. For the PEB in Fig 3(a) as an example, the instructions 6 to 8, which access the same cache block as the instruction 5, are not identified as access events. The reason is that only the first access of consecutive accesses to a same cache block could potentially cause a *miss* and restore the cache block and consequently the following accesses always hit.

Similarly, for *data access events*, we check the time points when memory load/store or I/O instructions are scheduled in their execution stages. For example, we assume that instruction 5 is a load instruction and hence identify a data access event when it is at the execution (EX) stage.

Finally, we complete the analysis for the PEB in Fig. 3(a), where a total of two instruction and one data access events are identified at their corresponding access time points (i.e., 0, 3, and 5). Since the start time of the analyzed PEB execution is unknown at static time, we denote these time points using the time offsets from the beginning clock cycle of the PEB.

B. Dynamic Timing Calculation

During simulation, the PSM issues the access events based on the pre-analyzed PEBs. For a currently executed basic block, a PEB is first selected according to runtime information such as branch prediction result or last executed basic block ID. Then the actual access time is calculated by summing up the statically analyzed time offsets of the access events in the selected PEB and the actual execution start time of this basic block, which is known after simulating the last executed block. Furthermore, access time points are adjusted according to whether the issued access events cause cache miss.

For instance, suppose that the branch prediction succeeds and basic block C is executed after basic block A during simulation. Through this, the PEB in Fig 2(d), whose access events are analyzed in Fig. 3(a), is selected. The actual access event time points are calculated by adding the pre-analyzed time offsets with the execution start time of this basic block (assume to be x for discussion) as shown in Fig. 3(b). Furthermore, assume the second access event of the PEB causes a cache miss during simulation and the pipeline is temporarily frozen for a three-cycle delay; accordingly, the third access is adjusted with an additional delay of three cycles (e.g., 5→8). The cache simulation will be discussed in section 5.

C. Discussions

The processor pipeline model we adopted here is assumed to be of in-order execution style. With this assumption, the cache delay can be easily incorporated during simulation without affecting the statically analyzed PEBs, because the in-order execution processor pipeline is frozen during cache miss [1]. In practice, despite this in-order assumption, this PSM is still extensively applicable since most embedded processors such as ARM 7 and ARM 9 are all of in-order execution style.

The time complexity of the static analysis is cn or O(n) where n is the number of basic blocks of the program analyzed and the constant c is the number of PEBs to be analyzed for each basic block. In practice, the constant c is typically small. With the fact that the average basic block size is about four to six [14], if we take as an example ARM 9, whose pipeline length is five, the corresponding c is about two or three. For this case, only instructions of about two consecutive basic blocks can be executed simul-

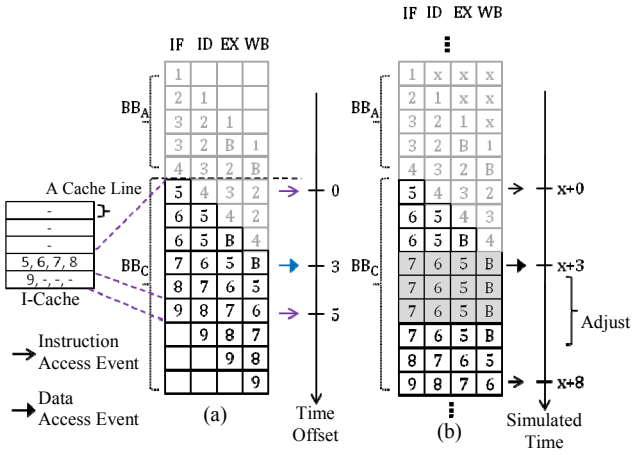

Figure 3: (a) An example of static analysis of access events in a PEB; (b) An example for dynamic timing calculation.

taneously in the pipeline.

V. CACHE SUBSYSTEM MODEL (CSM)

For accurate CCA processor modeling, a CSM should return correct access delay time to the access events issued from a PSM and trigger external accesses accurately on the processor interface. Therefore, the idea is to implement a model for each hierarchical cache in CSM such that it can return correct access delay values depending on hit/miss results. In addition, if a higher level cache is missed, the access request is passed on to the next cache hierarchy at correct timing. As a result, if all the cache hierarchies in a CSM behave correctly, access delays to the CSM can be calculated properly and all external accesses will be executed at accurate time points.

To model the timing behavior of each cache hierarchy, we adopt the CCA memory modeling method in [12]. Given a CA cache timing model, the method generates a corresponding computation tree, in which each path describes the timing of possible cache access behavior (such as miss and hit).

For illustration, Fig. 4(a) shows a processor with two hierarchical caches, L1 and L2. For clarity of discussion, we show only the clocked finite state machine (CFSM) which describes the cycle-by-cycle state transition behavior of the L1 cache. Upon an access request, L1's CFSM will perform hit/miss evaluation. Next, if the requested data is *hit*, the cache will return the requested data and stay in state *s0*; if not, the state will progress through *s1* to *s2* and start a handshaking process to request access with the next hierarchy until the assertion of signal *data_ok*, which notifies the completion of cache block restoring.

Then, the CFSM is converted into a compressed computation tree as in Fig. 4(b). The two paths of the computation tree correspond to the two types of the cache timing behaviors, i.e., *hit* and *miss*, for this particular case. The left path of the computation tree describes the *hit* case, which needs only one cycle for completion. The right path describes the *miss* case, which needs two cycles before and one cycle after an additional handshake with the next hierarchy.

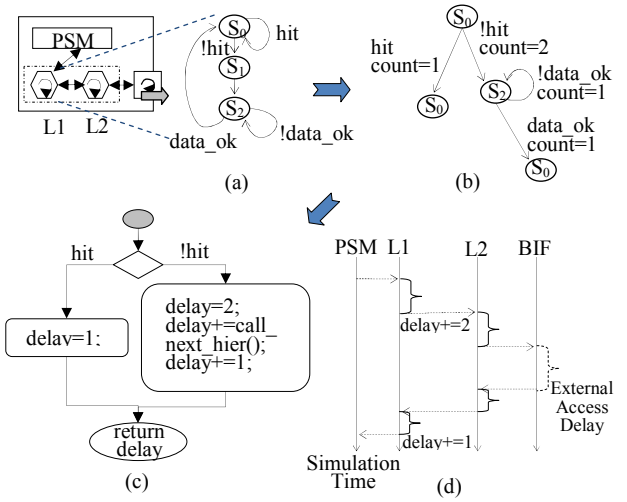

Figure 4: (a) A two-level cache system with a CA L1-cache model; (b) A CCA L1-cache model; (c) A procedure call implementing the CCA L1-cache model; (d) The sample simulation behavior.

Finally, the CCA cache model is implemented by a procedure call as in Fig. 4(c). Different paths in the computation tree are represented by different control flow branches. Access requests to the next hierarchy are implemented as function invocation to trigger actions in the next hierarchy.

Fig. 4(d) illustrates the CSM simulation behavior. Once the PSM requests an access to the CSM, the access is passed onto the L1 cache. Assume that the access causes a miss and consequently the L1 cache triggers an access to the next cache hierarchy after a two-cycle delay. Subsequently, if L2 also misses, it will trigger external memory access accurately according to its pre-analyzed timing. On the other hand, if the access is a hit in either L1 or L2, the procedure will return immediately with an accurate delay value.

Finally, a CCA processor model can be generated using the PSM and CSM. The following case study demonstrates the application of the proposed CCA model on an industrial-strength OpenRISC 1200 platform.

VI. A CASE STUDY

A. OR1200 CCA Processor Modeling

We chose an OpenRISC 1200 (OR1200) processor for testing our proposed approach because OR1200 is an open-source platform with all design details available. Hence, the generated models can be verified relatively easily.

OR1200 is a 5-stage 32-bit RISC processor in Harvard architecture. The processor is claimed to have been taped out in typical 0.18 um 6LM process, and it can provide over 150 Dhrystone or 2.1 MIPS performance at 150 MHz. The performance is reported to be comparable to competitors such as ARM9 processor [13].

The target architecture of OR1200 platform is shown in Fig. 5. To demonstrate the simulation performance gain of the CCA models, we use the fast compiled instruction-set simulation (ISS) technique from [15] for functional simulation.

B. Experimental Results

The experimental results are listed in Tab. I and most test cases are from OpenRISC official testbenches. Additionally, a 32-frame MPEG-4 QCIF video application is tested on the platform, where the processor fetches the encoded frames from the ROM for decoding and transfers the de-

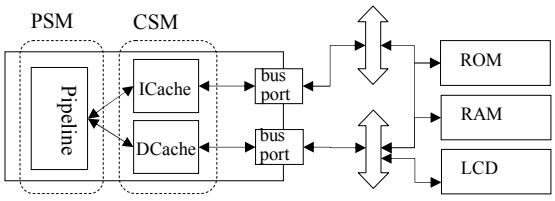

Figure 5: The modeled OR1200 platform.

Table I: Experimental Results

Test-case	# of BB	Anal. Time (s)	Trad. CA speed (mcps)	Comp. CA speed (mcps)	CCA speed (mcps)	Speed -up
fib	130	0.12	1.33	2.69	68.31	51X
mul	140	0.13	1.42	2.67	53.72	38X
cbasic	202	0.25	1.55	2.66	62.56	40X
dhry	236	0.33	1.94	2.88	117.12	60X
mpeg4	1370	2.60	1.93	2.87	114.51	59X

coded frames to the LCD for display.

For accuracy verification, the simulated clock times of bus accesses from the generated CCA processor model are checked against that of the target RTL model. Also, each test-case run on the generated CCA model has the same execution cycle count as on the RTL model.

Simulation speeds are shown in *million cycles per second* (MCPS) for comparison. The proposed model, *Compiled CCA,* is on average 50 times faster than the *Traditional CA* simulator, an interpretive ISS with a CA timing model. In comparison, C*ompiled CA*, which uses the compiled ISS technique with the CA timing model, is barely twice the speed of the *Traditional CA* approach. This shows that no significant simulation speed-up can be achieved when only using a fast ISS technique with the CA timing model, because the CA timing simulation contributes a great portion of simulation time.

The table also lists the pre-analysis time (*Anal. time*) of each test-case. It linearly increases as the number of basic blocks grows but is still negligible compared to the large simulation time. For example, the MPEG-4 case takes seconds for pre-analysis but minutes for simulation.

Finally, to demonstrate the performance evaluation capability of the CCA processor model, we run the MPEG-4 application with various configurations of different processor cache sizes. Fig. 6 shows the evaluation results, where the y-axis shows the execution time of the application on the platform, and each curved line is a result of a selected D-cache size along with various I-cache sizes, as indicated on the x-axis. Based on the results, D-cache sizes larger than 1KB and I-cache sizes larger than 4KB are not recommended, since no noticeable improvement is observed.

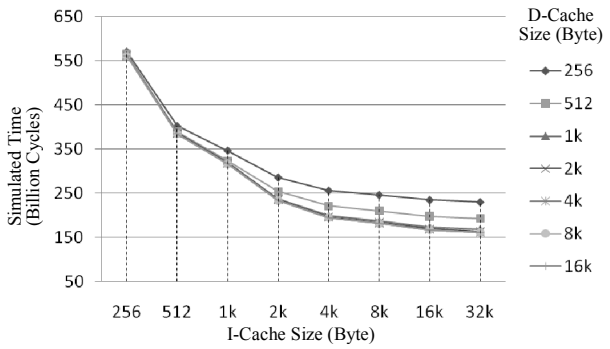

Figure 6: Performance evaluation for an MPEG application on OR1200 with different cache sizes.

While the CCA simulation finishes this evaluation process in 3 hours, the CA simulation for the same process takes more than 6 days.

VII. Conclusion

This paper introduces the concept of CCA processor modeling and proposes the first CCA processor modeling technique. The experiments show that the superior simulation speed and accuracy provided by the proposed approach greatly benefit the system design tasks.

For future work, we are currently investigating how to apply the CCA processor modeling concept to out-of-order execution type processors.

Reference

[1] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd ed., 2004.

[2] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the Transaction Level," in *Proc. of the conf. on Design Automation and Test in Europe*, pp. 3-8, 2008.

[3] J. Schnerr, O. Bringmann, A.Viehl, and W. Rosenstiel, " High-performance timing simulation of embedded software," in *Proc. of the Design Automation Conf.*, pp. 290-295, 2008.

[4] K. Lin, C. Lo, and R. Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," in *Proc. of the Asia and South Pacific Design Automation Conf.,*, pp.235-240, 2010.

[5] M. Wu, C. Fu, P. Wang, and R. Tsay, "An Effective Synchronization Approach for Fast and Accurate Multi-core Instruction-set Simulation," in *Proc. of the Conf. on Embedded Software*, pp. 197-204, 2009.

[6] L. Guerra, J. Fitzner, D. Talukdar, C. Schläger, B. Tabbara, and V. Zivojnovic, "Cycle and phase accurate dsp modeling and integration for hw/sw co-verification," in *Proc. of the Design Automation Conf.*, pp. 964-969, 1999.

[7] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "Lisa-machine description language for cycle-accurate models of programmable dsp architectures," in *Proc. of the Design Automation Conf.*, pp. 933-938, 1999.

[8] M. Reshadi, and N. Dutt, "Generic Pipelined Processor Modeling and High Performance Cycle-Accurate Simulator Generation," in *Proc. of the conf. on Design Automation and Test in Europe*, pp. 786-791, 2005.

[9] M. W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. A. Jerraya, "Debugging hw/sw interface for mpsoc: Video encoder system design case study," in *Proc. of the Design Automation Conf.*, pp. 908-913, 2004.

[10] S. Yoo, and K. Choi, "Optimistic distributed timed cosimulation based on thread simulation model," in *Proc. of the Workshop on Hardware/Software Codesign*, pp. 71-75, 1998.

[11] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration," in *Proc. of the Design Automation Conf.*, pp. 113-118, 2004.

[12] Y. Lo, M. Li, and R. Tsay, "Cycle count accurate memory modeling in system level design," in *Proc. of the conf. on Hardware/Software Codesign and System Synthesis*, pp. 287-294, 2009.

[13] OpenRISC, available on: http://www.opencores.org/openrisc,overview

[14] J. Huang and D. Lilja, "Exploiting Basic Block Value Locality with Block Reuse," in *Proc. of the Symp. on High Performance Computer Architecture*, pp. 106-115, 1999.

[15] M. Burtscher and I. Ganusov, "Automatic Synthesis of High-Speed Processor Simulators," in *Proc. of the Symp. on Microarchitecture,* pp. 55-66. 2004.

[16] S. Lim, et al. "An Accurate Worst Case Timing Analysis for RISC Processors," *IEEE Trans. Softw. Eng,* Vol. 21, Issue 7, pp. 593-604, 1995.