Fast approximately timed simulation

Vania Joloboff, Shenpeng Wang, Yangdong Deng

Fast approximately timed simulation

V. Joloboff[1], S.P. Wang[2], Y.D. Deng[2]
[1]*INRIA, East China Normal University, China and INRIA, France*
[2]*Tsinghua University, Beijing, China*

Abstract

In this paper we present a technique for fast approximately timed simulation of software within a virtual prototyping framework. Our method performs a static analysis of the program control flow graph to construct annotations of the simulated program, combined with dynamic performance information. The static analysis estimates execution time based on a target architecture model. The delays introduced by instruction fetch and data cache misses are evaluated dynamically. At the end of each block, static and dynamic information are combined with branch target prediction to compute the total execution time of the blocks. As a result, we can provide approximate performance estimates with a high simulation speed that is still usable for software developers.

Keywords: virtual prototyping, virtual platform, performance estimation, cycle-accurate simulation, approximately timed simulation, Instruction Set Simulation, ISS, dynamic binary translation, hot path, computer architecture, processor model, System-on-chi, pipelines, software.

Introduction

Performance of embedded software is important and must meet the requirements. Many embedded application developers use virtual prototyping framework including a hardware simulator to run the application software over a virtual platform. The simulation community has forged the terms Timeless, Loosely Timed (LT), Approximately Timed (AT) and Cycle Accurate (CA) to characterized simulators, although there is no formal definition of each category. Timeless simulators are purely functional and do not provide any notion of clock, LT simulators provide a notion of time, but the timing provided may be fairly inaccurate, AT simulators are supposed to provide performance estimates within a reasonable error margin and CA simulators (e.g. RTL level) are used to

simulate exactly digital circuits. There exists very fast Timeless or LT simulators such as BOCHS[1], or QEMU[2] or SimSoC[3] that can run the application software to validate its functionality and possibly test real time software using timers. But such simulators do not provide good enough timings to evaluate the software performance. On the other hand CA simulators are too slow to be used by application software developers. The idea of *Approximately Timed* simulation is to provide estimates that are as close as possible to the hardware performance, but at a simulation speed that is still an order of magnitude faster. An Instruction-Set Simulators (ISS) is typically used to emulate the behavior of the target on a host machine to execute embedded running on the target processor. In this paper we investigate an approach to provide a fast Approximately Timed ISS that provides good precision estimates. It consists in developing a higher abstraction model of the processor that still execute ISS instructions, but in parallel maintains some architecture state to measure the delays introduced by cache misses and pipe-line stalls, although the pipe-line is not really simulated.

Related work

There has been much work done around Instruction-Set Simulation to reach high simulation speed. Two popular instruments to describe higher abstraction level of hardware models are SystemC [4] and Transaction Level Modeling [5, 6], widely used in industry and academia and many efforts have been done to improve Instruction-Set-Simulator (ISS) speed. Whereas the early ISS's used interpretive simulation [7], most recent ISS's have used some kind of dynamic binary (cached) translation to accelerate simulation [8, 9, 10, 11]. Regarding performance estimate, given the slowness of CA simulation, attention has turned towards sampling. With sampling, only a few chunks of code sequences are selected and analyzed. Then statistical methods are used to generate performance estimates. Popular representatives of sampling methods are Simpoint [12], SMARTS [13] and EXPERT [14]. Sampling techniques can provide fairly accurate performance prediction with a high probability but may also generate very large data files. Another approach consists in statistical workload generation and simulation after collecting data from one complete simulation [15, 16]. Because cache misses are the major performance bottlenecks, specific attention has been devoted to cache performance and cache behavior prediction. For example, statistical methods have been successfully used for cache analysis in [17]. Yet another approach consists in doing compile time static analysis of the code for predicting cache behavior [18] based on cache miss equations. This method is very fast, but limited in the scope of programs that can be analyzed, and of course requires access to the source program.

More recently, the approach of code annotation has been used, with several variants and our method is affiliated to it as well. The idea consists in annotating the basic blocks of the code back with additional performance or power consumption data that is later used to compute non-functional properties. Such static data is obtained from a model of the target platform in some way. It can be obtained from intermediate compiler representation, or from the real target code

of the application [19, 20, 21]. The application code can then be either host compiled [22] using an abstract RTOS [23], together with the back annotation, or run into a virtual prototype, possibly combining both [24]. Purely static data cannot encompass dynamic behaviour of the hardware, such as cache and branch prediction. Our method consists in annotating the target code with information statically obtained from the binary target application code, based upon an abstract processor architectural model, and then execute that target code with an ISS to obtain the dynamic information. Moreover, we consider an approximation of that dynamic information in order to speed up simulation by introducing usage of hot paths and by constructing tables of bounded size (linear with code size) to store dynamically obtained information that will not need to be recomputed each time. When the table has reached its maximum size, its contents are used statically and are not recomputed.

Approximately timed simulation

Our method is using full system simulation of the target program, with an ISS. During the simulation, cache misses and hits are accurately detected and the control flow graph (CFG) in basic blocks of the program is constructed. A static analysis of the basic blocks is achieved for predicting their performance only once. After that, the simulator collects the performance of the individual basic block. Our work has started from the open source SimSoC Simulator [25] that includes SystemC using the TLM standard for communications among simulation models. SimSoC uses a dynamic translation schema whereby the code to be executed is analyzed and the Control Flow Graph of the simulated program is dynamically constructed. After some simulation time, a large part, if not all, of the application basic blocks are stored into the cache. The SimSoC simulator as released in open source is Loosely Timed, using a naïve approach: after executing *N* instructions, the clock is increased by the time of executing these instructions based on static configuration parameters. Our work has focused on creating an Approximately Timed version of the Power architecture ISS, leveraging off from the existing basic block structure.

Modern processors have complex architectures and can execute a certain number of instructions per clock cycle. There are however several cases where the instructions flow is disrupted, introducing delays in the computation. The major causes of delays are because (i) there are cache misses (data or instruction), or (ii) the pipe-line is stalled or (iii) there are wait states because of communication with peripherals. The latter delays can be captured by TLM transactions. In order to make an AT simulator, our idea is to simulate enough of the processes causing the processor delays, yet we are not simulating the exact hardware. We use instead a model with which the delays can be computed with good approximation while maintaining fast simulation. Our method consists in evaluating these delays with the following approach.

First, we approximate the delays created by the instruction cache misses. We have implemented a cache simulator that does not simulate the specific hardware cache in detail, but reproduces enough of the cache behavior to tell whether there

is a cache hit or miss. The instruction buffer is also simulated so that we can compute whether or not the pipe line is fed with instructions. Second, we have built a model of the architecture that makes it possible to evaluate delays without reproducing in detail the hardware. Third, we evaluate with a high precision only the most frequently executed code (*the hot blocks*), and use a lower precision for the code that is rarely used (*the cold blocks*), and finally we rely on TLM interface to obtain I/O delays. We perform a static analysis of basic blocks only once to construct the annotations to the code, assuming that future execution of these blocks will take the same number of cycles regarding the pipe line status. Only the data cache hit/misses need to be analyzed dynamically. On average, a program spends a lot of time to execute a small portion of its code, the hot paths. Since accurate performance estimate is costly, an idea to speed up AT simulation is to measure the time spent in infrequently used code (the *cold blocks*) with a low precision, but fast to obtain, and conversely measure with a high precision the *hot blocks*. The LT method is used for a block until it exceeds some threshold, then performance estimation switches to the accurate method. This introduces an additional error margin but accelerates simulation and the higher error margin introduced on cold blocks is acceptable. As the threshold is a configurable parameter, the developers can always obtain more accurate simulation by setting the threshold to 1.

The main SimSoC simulator structure is maintained in the AT version. The ISS still executes instructions on the basis of basic blocks and all of the instructions of a block are executed in one step, but at the same time an annotation is constructed. The instructions can be categorized into three categories (a) those that may not stall the pipeline, or those that may, either because they (b) have a dependency on missing data, or (c) influence the branch prediction, which can only be the last instruction of the basic blocks. Therefore we can perform a static analysis of a block to know where the pipe line would stall and compute the delays, to annotate the block with this information. For example, if the pipe-line has K-issues specialized for some instructions, it is necessary to watch for the execution of the basic block once to see which instructions will be parallelized and which instructions will inevitably stall. This simulation does not need to reflect the full pipe line architecture, it is sufficient to understand the dispatching of instructions into the pipe line issues and the stalling factors. After one execution of a basic block, the timing is known and an annotation is constructed.

Given the instruction cache size and cache line size of the target architecture, one can to know whether there is a pre-fetch miss in a block. The simulator checks at the entrance of each basic block whether or not there will be a fetch miss inside that block. As many blocks are small compared to the (instruction) cache line, there may be none. If there is a fetch miss, it is still necessary to simulate the Instruction Buffer process as it has the effect to reduce cache miss delays. Only if the instruction buffer becomes empty while the cache is refilled, then it results into a delay, but in many cases, the instruction buffer process is compensating for the cache miss. This is architecture dependent, as the instruction buffer may be filled at different rates but this can be configured with

parameters. Simulating the interconnect behaviour to compute the fetch miss would slow down simulation. However, one can approximate that a delay is constant for all occurrences of a cache miss at a particular position in the basic block. Hence, in long loops over a series of basic blocks, it is not necessary to re-compute the delay each time. One can maintain for a basic block a table of fetch miss delays that gets computed during the first occurrences of the loop, and later re-used as constants. This table has a maximum size of the size of the basic block... Note also that when a new cache line of N instructions is entered, then it is not necessary to check for a fetch miss for the following N-1 instructions as it is known to execute sequential instructions that already are in the cache.

A question is the state of the instruction buffer at the entrance of a basic block. There are several possibilities, depending upon the branch prediction algorithm. At the end of the previous basic block, a branch was taken or not. If the branch prediction was wrong, the instruction buffer has been flushed. If the branch prediction was correct, some architecture would have filled correctly the instruction buffer, some others would flush anyway. In this work, we have not considered modelling entirely the branch target prediction and we make the pessimistic assumption that the instruction buffer is empty at the entrance of the basic block. Our performance estimate in this case is worse than reality, which is useful for worse case analysis. In practice, it does not add a considerable error margin as this error only occurs when the branch prediction for not taken was correct and there is a cache miss at the beginning of the block.

Many modern processors, including our case study processor, have a branch prediction unit. These units serve two purposes, to calculate the effective address of the branch in advance, and to fill the cache ahead of time with the appropriate instructions. During simulation, the address of next instruction after the block is known. Thus, one can check, using a model of the processor branch target prediction, whether the predicted target is correct or not, and then possibly compute how many cycles the branch instruction will take.

Regarding the data cache, it must remain fully dynamic as the cache behavior is dynamic, and the application itself may dynamically reset cache options. The SimSoC ISS knows from the target address map whether the address is a real memory access or a mapped I/O. When it is real memory, it uses the Direct Memory Interface (DMI) of the TLM standard to accelerate memory access. We have added a cache algorithm that does not perform actual data caching, but emulates enough of the cache behavior to compute whether there is a cache hit or miss, in cache levels 1 or 2. Again, a pessimistic constant time delay is used for a cache hit or miss at each level, so that the ISS does not need to issue memory transactions and can keep using the DMI interface.

Validation and performance measurements

For this experiment we have considered an embedded processor, namely a Power Architecture EZ model. This processor has a pre-fetch instruction cache, a two issues pipe line, a branch prediction unit, but the pipe line has a small number of stages. The simulation is driven by the SimSoC framework. We have not been

able to run the SPEC benchmark programs used by other performance prediction reports because these benchmarks need OS support to construct reporting files not supported by SimSoC Embedded Linux simulator. We can however run open source benchmarks like the libcrypto cryptographic library and various sorting algorithms. Also we have constructed specific small tests to test special cases, that our partner ST Microelectronics could run on their cycle accurate simulator or real hardware, to help us validate our model. As a result, our current system generates in general accurate performance estimation within the error that we had set as our own goal. We have one error due to the fact that, in the current implementation as of this writing, we have not modelled yet the store buffer and this error shows up clearly in the specific benchmarks for store instructions. The impact of running the performance models in addition to the full system simulation has degraded performance, which was expected, but the performance loss is an average of about 33%.

Conclusion

We have presented here a Fast Approximately Timed method of embedded systems simulation. We have added to an existing ISS an abstract model of a processor so that it annotates each basic block with the number of cycles it will take to execute it. This model includes a model of the pre-fetch system, a model of the pipe line and a model of branch prediction. To construct the annotation; it executes the block semantics in one short step, and simultaneously evaluates the time to execute that block. On purpose, for the sake of simulation speed, our model is not totally accurate. It purportedly makes approximations by using constant values instead of issuing transactions to obtain a realistic and it does not maintain micro architecture state across basic blocks. In addition, we have two models, a simplified model and an approximate model. The simplified model is used for cold paths in the code and the more complex, slower, approximate model is used only for hot paths. We obtain an accurate timing only for the most frequently executed code, but this code accounts for a large part of the application timing. Because it uses static analysis of basic blocks to construct the annotation, which is done a limited number of times, it results in reasonably fast simulation, usable by software developers. We have measured our method to be effective to simulate a Power EZ model. Resulting performance estimates are within expectation. This method has the advantage of simplicity; no other external tool is required. The performance estimation is delivered at the end of the simulation session. The same tool can be used by software developers to validate the functionality of application code within the full system simulation of SimSoC framework and obtain performance estimates.

Acknowledgements

We thank ST Microelectronics for their support in this project for comparing simulation results with cycle accurate platforms.

References

[1] Lawton K.P., “Bochs: A Portable PC Emulator for Unix/X”, in *Linux Journal, Volume 1996, Issue 29es, Article No. 7,* 1996

[2] Bellard F., “QEMU, a Fast and Portable Dynamic Translator”, in the *USENIX Annual Technical Conference,* Anaheim, CA, USA, p. 41-46, 2005.

[3] Helmstetter C., Joloboff V., "SimSoC: A SystemC/TLM integrated ISS for full system simulation," in *IEEE Asia Pacific Conference on Circuits and Systems*, APCCAS 2008, Macau, China, 2008.

[4] SystemC Language Reference Manual (IEEE Std 1666-2005), Open SystemC Initiative,http://www.systemc.org/, 2005.

[5] Cai L., Gajski D., “Transaction level modeling: an overview,” in the *1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS ’03. New York, NY, USA: ACM, 2003.

[6] Donlin A., “Transaction level modeling: flows and use models,” in *the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS ’04 New York, NY, 2004.

[7] Sutarwala S., Paulin P.G., Kumar .Y, “Insulin: An instruction set simulation environment,” in *CHDL ’93: 11th IFIP WG10.2 International Conference.* Amsterdam, North-Holland Publishing Co., pp. 369–376, 1993

[8] Reshadi M. , Mishra P. , and Dutt N. “Instruction set compiled simulation: a technique for fast and flexible instruction set simulation,” in *Design Automation Conference, DAC'03*, pp. 758–763, 2003.

[9] Nohl A., Braun G., Schliebusch O., Leupers R., Meyr H. and Hoffmann A., “A universal technique for fast and flexible instruction-set architecture simulation,” in *the 39th conference on Design automation, DAC’02.* New York, USA, pp. 22–27, 2002.

[10] Scott K., Kumar N., Velusamy S., Childers B., Davidson J.W. and Soffa M.L., “Retargetable and reconfigurable software dynamic translation,” in *International Symposium on Code Generation and Optimization,* CGOAZ´03, 2003.

[11] Shi H., Wang Y., Guan H., and Liang A., “An intermediate language level optimization framework for dynamic binary translation,” *SIGPLAN Notices, vol. 42, no. 5*, pp. 3–9, 2007.

[12] Hamerly G., Perelman E., and Calder B.,“How to use simpoint to pick simulation points,” *SIGMETRICS Perf.. Eval. Rev., vol. 31, no. 4,* pp. 25–30, 2004.

[13] Wunderlich R., Wenisch T., Falsafi B., and Hoe J., “Smarts: accelerating microarchitecture simulation via rigorous statistical sampling” in proceedings *30th Annual International Symposium on Computer Architecture*, ISCA'03, 2003.

[14] W. Liu and M. C. Huang, “Expert: expedited simulation exploiting program behavior repetition,” in *the 18th annual international conference on Supercomputing*, ICS ’04. New York, USA., 2004.

[15] Nussbaum S. and Smith J. E., "Modeling superscalar processors via statistical simulation," in *International Conference on Parallel Architectures and Compilation Techniques*, pp. 15–24, 2001.

[16] Rao R., Oskin M. and Chong F., "Hlspower: Hybrid statistical modelling of the superscalar power-performance design space," in Procs. Of *High Performance Computing, HiPC 2002*, LNCS, Springer, vol. 2552, pp. 620–629, 2002.

[17] Berg E., Zeffer H., and Hagersten E., "A statistical multiprocessor cache model," in *International Symposium on Performance Analysis of Systems and Software*, pp. 89–99, 2006.

[18] Vera X. and Xue J., "Let's study whole-program cache behaviour analytically," in Proceedings of the *8th International Symposium on High-Performance Computer Architecture, HPCA'02,* Wash. DC, USA, 2002.

[19] Hwang Y., Abdi S., and Gajski D., "Cycle-approximate retargetable performance estimation at the transaction level," in the *Conference on Design, Automation and Test in Europe, DATE'08*. NY, USA, pp. 3–8, 2008.

[20] Bouchhima A., Gerin P., and Petrot F., "Automatic instrumentation of embedded software for high level hardware/software co-simulation," in *Asia and South Pacific Design Automation Conference*, pp. 546–551, 2009.

[21] Stattelmann S., Bringmann O., and Rosenstiel W., "Fast and accurate source-level simulation of software timing considering complex code optimizations," in Proc. of the *48th Design Automation Conference, DAC '11*. New York, NY, USA, ACM,, pp. 486–491, 2011.

[22] Gerstlauer A., Chakravarty S., Kathuria M., and Razaghi P., "Abstract system-level models for early performance and power exploration," in *Asia and South Pacific Design Automation Conference*, pp. 213–218, 2012.

[23] Krause M., Englert D., Bringmann O., and Rosenstiel W., "Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation," in t*he 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '08, NY, USA, pp. 143–148, 2008.

[24] Gao L.,Karuri K., Kraemer S., Leupers R., Ascheid G., and Meyr H., "Multiprocessor performance estimation using hybrid simulation," in Design Automation Conference, DAC 2008, 2008.

[25] INRIA Gforge, "Simsoc open source software." Available: http://gforge.inria.fr/projects/simsoc.