Automated, Retargetable Back-Annotation for Host Compiled Performance and Power Modeling

Suhas Chakravarty, Zhuoran Zhao, Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin
{suhas.chakravarty,zhuoran,gerstl}@utexas.edu

ABSTRACT

With traditional cycle-accurate or instruction-set simulations of processors often being too slow, host-compiled or source-level software execution approaches have recently become popular. Such high-level simulations can achieve order of magnitude speedups, but approaches that can achieve highly accurate characterization of both power and performance metrics are lacking. In this paper, we propose a novel host-compiled simulation approach that provides close to cycle-accurate estimation of energy and timing metrics in a retargetable manner, using flexible, architecture description language (ADL) based reference models. Our automated flow considers typical front- and back-end optimizations by working at the compiler-generated intermediate representation (IR). Path-dependent execution effects are accurately captured through pairwise characterization and back-annotation of basic code blocks with all possible predecessors. Results from applying our approach to PowerPC targets running various benchmark suites show that close to native average speeds of 2000 MIPS at more than 98% timing and energy accuracy can be achieved.

1. INTRODUCTION

Software developers typically rely on executable models for quick and accurate feedback on the performance and power of their designs. Traditionally, cycle-accurate instruction set simulators (ISSs), micro-architectural or RTL/gate-level descriptions have been used to perform performance and power simulations of applications executing on a processor core. Their drawback is that they are either inaccurate or slow, since they require the processor micro-architecture either to be fully abstracted or to be modeled in detail.

As an alternative to ISS-based models, high-level software and processor models based on native, so-called host-compiled or source-level software execution have recently emerged. Such approaches model computation at the source code level (typically in C-based form), which allows a purely functional model to be natively compiled onto the host for fastest possible execution. Timing and power information is added by prior back-annotation of the source with estimated target metrics. In complete host-compiled models, annotated source code is then further wrapped into models of operating systems and processors that integrate into standard transaction-level modeling (TLM) backplanes.

Previous host-compiled approaches have thus far mostly focused on timing simulations. Furthermore, they are often tied to specific target architectures and limited in their accuracy or speed of capturing basic path-dependent micro-architectural execution effects. The main contribution of this paper is to propose a fast and accurate host-compiled simulation approach for automated and retargetable modeling of both performance and power consumption. Our flow is built by annotating the compiler generated intermediate representation (IR) of the application source code with estimates obtained from reference timing and energy models. Working at the IR level allows us to accurately trace execution paths during simulation, where we establish a mapping from the binary control flow graph (CFG) to the IR such that compiler backend optimizations are fully considered. We leverage existing, open-source architecture description language (ADL) frameworks for cycle-accurate timing and energy characterization across a wide range of targets.

We further apply a pairwise characterization of each basic code block with all possible predecessors to accurately capture path-, state- and pipeline-dependent effects. Automated one-time back-annotation of code is fast (on the order of 1-2 minutes), while resulting models are shown to simulate at close to source-level speeds (of more than 2000 MIPS on average) with near cycle accuracy (less than 0.8% average timing and energy error).

1.1 Related Work

There is a range of approaches that aim to annotate timing information obtained from a target model back into application code either directly at the source [1–4] or at the intermediate representation [5–8]. A problem with working at the source level is that it can result in inherent inaccuracies in the mapping between target and source code under aggressive control flow optimizations. To resolve ambiguities, most approaches either fall back to [1] or establish a separate path-tracking [2] via an IR-level simulation model. We avoid these issues by working at the IR directly. Nevertheless, in the presence of aggressive compiler optimizations, even IR and binary control flows do not always match. Existing approaches either disable optimizations and rely on debug information [5], or obtain high-level estimates directly from the IR [6] or source code [4]. We have found debug information of optimized code to be unreliable. We therefore implement an approach that combines a flow graph matching algorithm with debug information as fall back.

For accuracy, we perform back-annotation using cycle-accurate simulation of actual target binaries at the level of basic blocks. Other binary-based approaches instead rely on static code analysis [1, 2, 5, 8], which is often overly conservative and tied to a specific backend target. By contrast, our approach is designed to be accurate and fully retar-

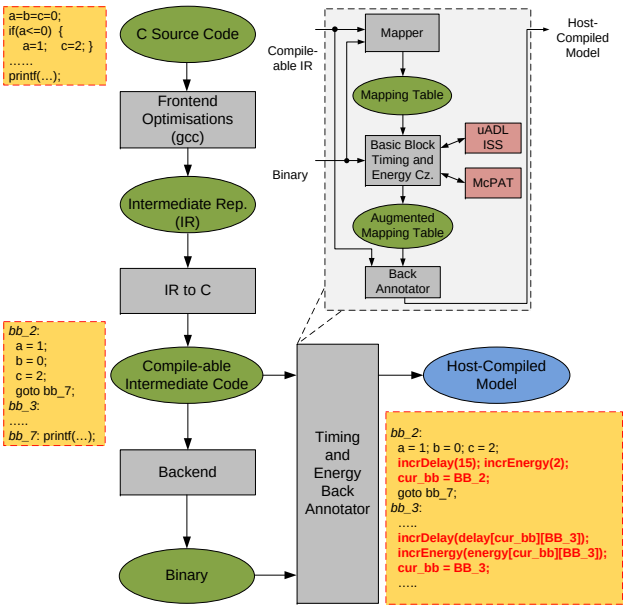

Figure 1: Host-compiled back-annotation flow [15].

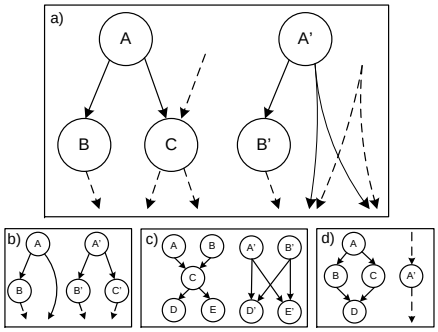

Figure 2: Mapping of IR to binary control flows.

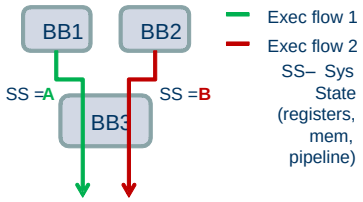

Figure 3: Path dependency of processor state.

getable. Furthermore, since off-line characterization is only performed once per static block pair, it is fast while being able to take inter block timing into account. The work in [9, 10] relies on a similar approach for path-dependent characterization of timing. However, they do not include power estimation and are applied to relatively slow instruction-set simulation or abstract pipeline models, neither of which guarantee accurate characterization of all blocks.

Existing power estimation approaches at the source or intermediate levels employ coarse-grain models that assume a constant or statistical energy consumption model at the granularity of complete instructions or source-level operations [11–13]. They thus largely focus on predicting the execution time correctly to arrive at an estimate of overall power consumed. By contrast, we leverage existing low-level reference models that operate at detailed micro-architectural granularity and make no such assumptions. By following a pairwise block characterization approach, we are able to maintain the accuracy of such models while achieving fast estimation and simulation times.

1.2 Flow Overview

Figure 1 shows our flow for automatic timing and energy back-annotation of host-compiled models. The application C code is passed through a generic cross-compiler front end (*gcc* in our case) to produce an IR, which is then further massaged back into compileable C form [14].

During following back annotation, the IR's CFG is then further augmented with timing and energy information. The IR is first passed to the generic cross-compiler backend of the chosen target processor (again, *gcc* in our case). The generated binary is then analyzed to extract its CFG and establish the mapping between basic blocks in the IR and binary. This mapping is needed to accurately determine annotation points in the IR. Basic blocks in the binary are then characterized for timing and energy consumption. This is done by executing them pairwise on a retargetable, cycle-accurate ISS, which is automatically generated from an open-source ADL infrastructure [16]. Execution statistics from the simulation are further fed to a retargetable, McPAT-based reference power model of the chosen processor [17]. The resulting timing and energy estimates per basic block are then back-annotated into the compileable IR, aided by the mapping. This final step creates the host-compiled model.

2. BACK-ANNOTATION

In the following, we describe the timing and energy back annotation process at the core of our flow.

2.1 Mapping

After constructing the CFGs, a valid graph mapping needs to be established in the presence of variations between the two graphs due to compiler backend optimizations. Typical mismatches between IR and binary CFGs are illustrated in Figure 2. In the context of back-annotation, a key criterion for a match to be valid is that the number of execution paths traversed and passing through both nodes during program execution has to be equal. We therefore perform a synchronized depth-first traversal of both CFGs to identify legal matches based on a control flow representation using both loop and branch nesting levels. GDB's mapping information is used when multiple equally likely matches are possible, as is the case in branches of if-then-else statements.

2.2 Basic Block Characterization

Accurate characterization is complicated by the fact that the timing and energy consumption of a basic block can be significantly affected by pipeline effects, which depend on the state of the processor at the start of block the execution. Hence the timing and energy consumption for a block is determined by code that has previously executed (Figure 3).

To approximate this effect, we characterize each block through pairwise executions with all of its possible predecessors. As experiments will show, a two-level characterization results in only a slight loss in accuracy in a few cases.

The presence of function calls in a basic block presents additional considerations. In the same way as regular blocks, during real execution the caller and callee can influence each

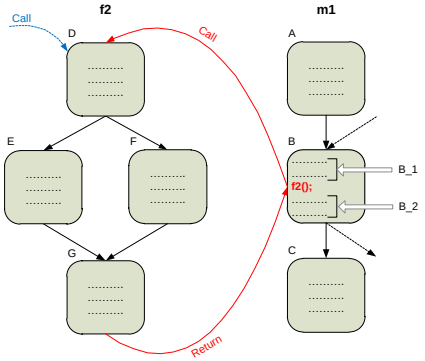

Figure 4: Function call characterization.

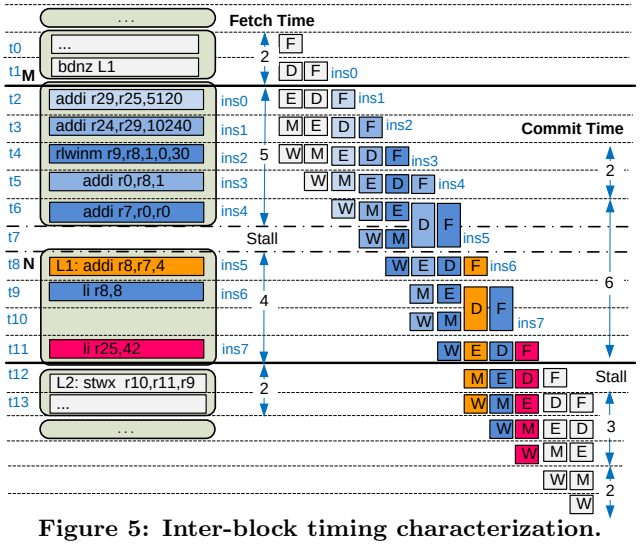

Figure 5: Inter-block timing characterization.

other's timing. Figure 4 shows a CFG snippet for a hypothetical application example. Consider the execution sequence A, B, D, F, G, B, C. The total execution time along this path is given by:

$$T_{Path} = T_A + T_{B\_1|A} + T_{D|B\_1} + T_{F|D} + T_{G|F} + T_{B\_2|G} + T_{C|B\_2},$$

where $T_{i|j}$ denotes the execution time of basic block i given that its immediate predecessor was block j. The equation shows that timing-wise, B is divided into smaller sub-blocks at the point of the function call to $f2()$, where the caller and callee's timings $T_{D|B\_1}$ and $T_{B\_2|G}$ are influenced by each other. Specifically, the processor state upon entry into a function depends on where it is called from. Similarly, state upon return to the callee depends on the last executed block in the called function. To capture this interdependencies, caller and callee are characterized in conjunction with each other. In our flow, blocks are split into sub-blocks at the point of each function call. A function's root basic block is characterized for each point of call, by pairing it up successively with all preceding sub-blocks. Likewise, the sub-block following a function call is characterized with all of the callee's exit blocks containing an implicit or explicit return statement.

2.2.1 *Timing Characterization*

To calculate the execution time of a basic block for a certain predecessor, the detailed trace generated from its pairwise ISS execution is analyzed (Figure 5). We rely on the difference in the fetch time instants of the first and last instructions in a characterized basic block to determine its execution time. Given a sequence of overlapping executions of successive blocks, annotating fetch delays is equivalent to recording commit times. Any stall inside the processor pipeline will propagate to successive instructions and eventually manifest itself both as delay in the stalled instruction's commit time as well as an equivalent delay in the fetch time of a following instruction.

If any stall occurs between fetches of different blocks, the execution time needs to be further adjusted to account for the gap with respect to the predecessor's end of execution. In Figure 5, the stall penalty for the fetching stage of the first instruction in basic block N is simply added to N's characterized execution time. By contrast, in multi-issue microarchitectures, the amount of overlap is subtracted from the execution time of the characterized block.

Table 1: Benchmark summary.

Benchmark	Suite	Modifications	Sim. instr.
SHA (Sm)	Security	Disabled tail-call	14,674,926
SHA (Lg)			153,067,895
ADPCM (Sm)	Telecom	-	36,658,548
ADPCM (Lg)			724,729,999
CRC32 (Sm)	Telecom	Disable inlining	13,688,752
CRC32 (Lg)			266,112,122
Sieve	-	-	12,284,657

2.2.2 *Energy Characterization*

Execution statistics such as instruction counts, types and functional units accessed are extracted from the detailed trace emitted during the ISS execution and further passed into McPAT. Combining these statistics with the block's characterized execution timing and cycles, the power consumption output from McPAT is then converted into an energy consumption figure for the block pair.

2.3 Back Annotation

The metrics gathered during the characterization step are recorded in the mapping table. As the last step in our flow, the mapping table is used for directing the annotation of target metrics into the compileable IR at the correct points.

The IR annotations are in the form of time and energy counters, a global array containing delay and energy estimates for all possible basic block pairs, and corresponding table lookups in each block to increment counters based on the ID of the current block and its runtime predecessor. In case of additional basic blocks in the binary, the annotation of the timing data for such a *bridging block* in the IR is done in the corresponding branch of the if-statement of its predecessor.

3. EXPERIMENTS AND RESULTS

We have implemented our flow in Python and applied it a subset of benchmarks from the MiBench suite. Back annotations and simulations were performed on a quad-core Intel i7 workstation running at 2.6 GHz. The uADL ISS [16] and McPAT [17] were used as timing and energy references, respectively.

Table 1 shows a summary of the benchmarks and the modification made to the original code or compilation process. The targets we evaluated are a PowerPC in-order, e200_z4-

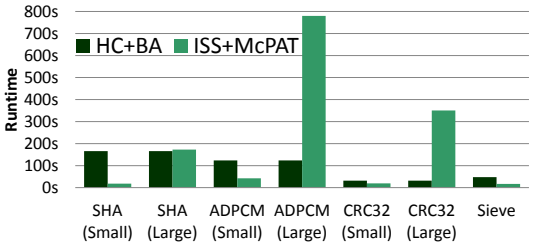

Figure 6: Runtime Comparison.

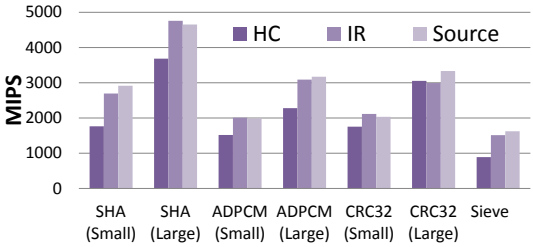

Figure 7: Simulation speed.

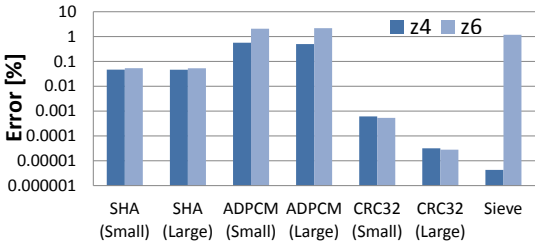

Figure 8: Host-compiled timing accuracy.

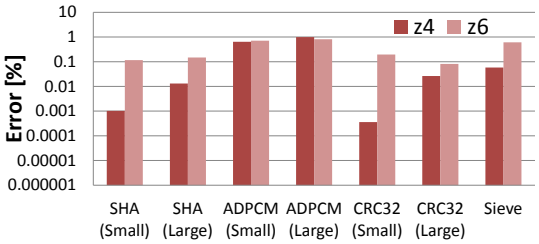

Figure 9: Host-compiled energy accuracy.

like dual-issue and a e200_z6-like single-issue core with no cache, MMU, floating point unit or dynamic branch predictor. A gcc-4.4.5 cross-compiler was used to generate code.

3.1 Speed Results

Figure 6 shows average runtimes of the back-annotation (BA) process for generation of final host-compiled simulation models from the original C source code. Simulation times of host-compiled models are compared against ISS and McPAT runtimes in a traditional simulation setup. As results show, for short simulations, the benefits of faster host-compiled simulations do not outweigh the overhead introduced by the extra back-annotation process. However, as input data sets and simulation lengths grow, combined back-annotation and HC runtimes increase at a much slower rate than total ISS plus McPAT times.

Figure 7 shows the simulation speeds of the host-compiled (HC) models as compared to that of source and IR level simulations. As is evident, there is no degradation in simulation speed by working with the IR instead of the source. On average, the host-compiled models are nearly 2100 times faster than the ISS running at an average of around 2100 MIPS.

3.2 Timing and Energy Results

Resulting timing and energy errors are summarized in Figure 8 and Figure 9. For the z4 target, the maximum estimation errors in timing and energy are 0.6% and 1.0%, respectively, while average errors are 0.2% and 0.3%. On the z6 target, the maximum timing and energy errors are 2.2% and 0.8% with an average error of 0.8% and 0.4%, respectively. Overall, results confirm that pairwise characterization represents a good tradeoff between significantly improved accuracy and low back-annotation runtime.

4. SUMMARY AND CONCLUSIONS

In this paper, we presented a framework for combined, fast and accurate power and performance estimation in a host-compiled simulation approach. Concerns of compiler optimizations are addressed by working at the IR level. The approach is fully retargetable by virtue of utilizing a standard ADL-based backend tool chain for timing and energy estimation. Our flow and its subsequent automation was evaluated on several industry-standard benchmarks executing on PowerPC targets. Results show order of magnitude speedups yet high accuracy in source-level simulation as compared to a cycle accurate ISS.

Future work will include development of data-dependent power models, integration with existing cache, OS and processor modeling approaches, and evaluation on a wider range of target platforms, micro-architectures and benchmarks.

5. REFERENCES

[1] Z. Wang and J. Henkel, "Accurate source-level simulation of embedded software with respect to compiler optimizations," in *DATE*, 2012.

[2] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *DAC*, 2011.

[3] H. Zabel and W. Müller, "An efficient time annotation technique in abstract RTOS simulations for multiprocessor task migration," in *DIPES*, 2008.

[4] C. Brandolese *et al.*, "Source-level execution time estimation of C programs," in *CODES*, 2001.

[5] Z. Wang and A. Herkersdorf, "An efficient approach for system-level timing simulation of compiler-optimized embedded software," in *DAC*, 2009.

[6] E. Y. Hwang, S. Abdi, and D. Gajski, "Cycle approximate retargettable performance estimation at the transaction level," in *DATE*, 2008.

[7] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic instrumentation of embedded software for high level hardware/software co-simulation," in *ASP-DAC*, 2009.

[8] J. Schnerr *et al.*, "High performance timing simulation of embedded software," in *DAC*, 2008.

[9] R. Plyaskin and A. Herkersdorf, "Context-aware compiled simulation of out-of-order processor behavior based on atomic traces," in *VLSI-SoC*, 2011.

[10] K.-L. Lin, C.-K. Lo, and R.-S. Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," in *ASP-DAC*, 2010.

[11] C. Brandolese *et al.*, "A multi-level strategy for software power estimation," in *ISSS*, 2000.

[12] C. Brandolese, S. Corbetta, and W. Fornaciari, "Software energy estimates based on statistical characterisation of intermediate compilation code," in *ISLPED*, 2011.

[13] D. Calvo *et al.*, "A multi-processing systems-on-chip native simulation framework for power and thermal-aware design," *Journal of Low Power Electronics*, vol. 7, no. 1, pp. 2–16, 2011.

[14] A. Goswami and A. Gerstlauer, "ExtractCFG: A framework to enable accurate timing back annotation of c language source code," CERC, UT Austin, Tech. Rep. UT-CERC-11-02, Aug. 2011.

[15] A. Gerstlauer *et al.*, "Abstract system-level models for early performance and power exploration," in *ASP-DAC*, 2012.

[16] Freescale, "ADL Release 2.0.0," http://opensource.freescale.com/fsl-oss-projects.

[17] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.