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.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development

Keywords

Power and performance modeling, Host-compiled simulation

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. The flow is fully automated and easily retargetable. We leverage existing, open-source architecture description language (ADL) frameworks for cycle-accurate timing and energy characterization across a wide range of targets. 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 further improve accuracy over existing approaches by relying on a pairwise characterization of each basic code block with all possible predecessors, both within and across the function hierarchy. This allows us to accurately capture path-, state- and pipeline-dependent effects that can significantly influence the dynamic execution behavior of each block of code. 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).

The rest of the paper is organized as follows: After an overview of related work and the back-annotation flow in the following sections, details of the timing and energy back annotation process will be described in Section 2. Section 3 then discusses the results of our experiments and Section 4 presents the conclusions.

1.1 Related Work

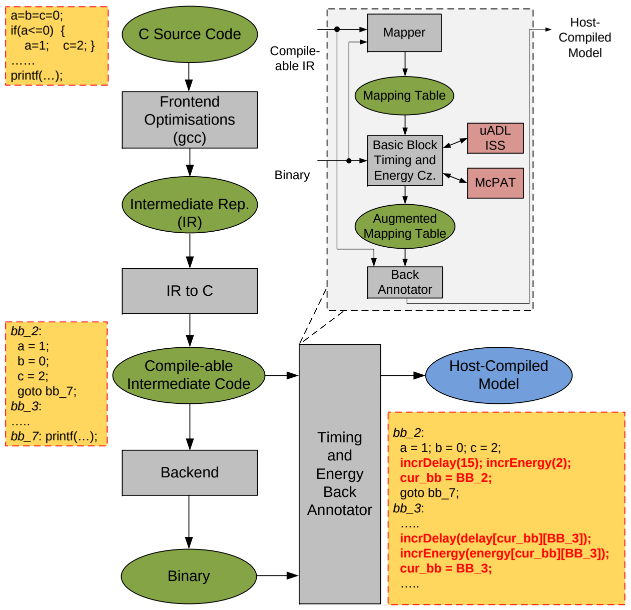
There is a range of approaches that aim to annotate timing information obtained from a target model back into application code either directly at the source [26,29,32] or at the intermediate representation [1,14,25,30]. A problem with working at the source level is that it can result in inherent inaccuracies in the mapping between the target and source code under aggressive control flow optimizations. To resolve ambiguities, most approaches either fall back to [29] or establish a separate path-tracking [26] 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 [30], or obtain high-level estimates directly from the IR [14] or source code [4,6]. 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 only when needed. A similar graph matching flow targeted at binary-to-source mapping is described in [19].

For accuracy, we perform back-annotation using cycle-accurate simulation at the level of actual target binaries. Other binary-based approaches instead rely on static code analysis [25,26,29,30], which is often overly conservative and tied to a specific backend target. By contrast, our approach is designed to be accurate and fully retargetable. Furthermore, since off-line characterization is only performed once per static block pair, it is fast while being able to take inter basic block timing into account. The work in [17,22] relies on a similar approach for path-dependent characterization of basic block 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.

To further capture dynamic effects, several approaches include dedicated simulation models of micro-architecture features such as caches or branch predictors [25,30] or of complete OS and processor models that include effects of task interleavings or other exceptions [12,24]. Furthermore, there are hybrid models that toggle between host-compiled and ISS-based execution [15,20]. Our approach is orthogonal to and supports integration with such dynamic models.

For power estimation, popular approaches are to leverage high-level state-based models, to develop macro models for micro-architectural functional blocks [8,27], to use detailed simulation-based modeling frameworks [5,16], or to support estimation at a range of abstraction [21]. Simulation-based methods utilize activity information gathered from cycle accurate performance simulators to estimate power consumption. Tiwari et al. [28] present an instruction level power model that also takes into account dynamic inter-instruction effects like pipeline stalls. However, their methodology is not portable in that it requires detailed profiling of the instruction set of the target. The authors in [13] present an approach wherein phases are identified in the program based on dynamic power consumption. Detailed power characterization of a representative slice from each phase is used to build a fast power simulation model.

At the source or intermediate levels, existing power estimation approaches employ coarse-grain models that assume a constant or statistical energy consumption model at the granularity of complete instructions or source-level operations [2,3,7]. 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.



```
a=b=c=0;
if(a<=0)  {
    a=1;    c=2; }
......
printf(...);
```

```
bb_2:
  a = 1;
  b = 0;
  c = 2;
  goto bb_7;
bb_3:
.....
bb_7: printf(...);
```

```
bb_2:
  a = 1; b = 0; c = 2;
  incrDelay(15); incrEnergy(2);
  cur_bb = BB_2;
  goto bb_7;
bb_3:
  .....
  incrDelay(delay[cur_bb][BB_3]);
  incrEnergy(energy[cur_bb][BB_3]);
  cur_bb = BB_3;
  .....
```

Figure 1: Host-compiled back-annotation flow [10].

An overview of our basic flow for retargetable power and performance back-annotation was first introduced in [10]. However, in earlier work, the flow was not automated, compiler optimizations were not fully supported and no details about the pairwise block characterization were presented. In this paper, we extend the original concept with a binary-to-IR mapping algorithm and accurate block characterization to provide a fully automated flow that supports complex optimized code.

1.2 Flow Overview

Figure 1 shows our flow for automatic timing and energy back-annotation of host-compiled models, accompanied by representative code snippets at various stages. 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 [11]. Working at the IR allows typical compiler front-end optimizations to be taken into account with little or no penalty in execution speed. For debugging, IR code can be annotated with source line information and thus linked back to the original application code. Moreover, the IR allows us to accurately observe effects of target-dependent behavior, such as overflows in the original C code. For this, the IR-to-C conversion script maps all variables and constants into a host data type of target-equivalent size and alignment. Furthermore, the IR inherently provides a close representation of the final control flow graph (CFG) of the target code and hence is able to accurately reflect all data-dependent execution behavior.

During following back annotation, the IR's CFG is then further augmented with timing and energy information. The

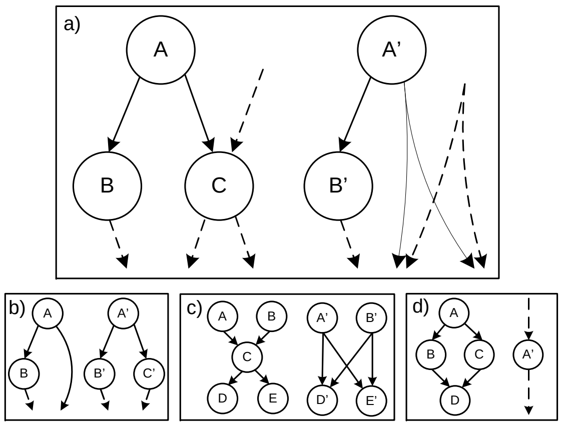


Figure 2: Mapping of IR to binary control flows.

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 (BBs) 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 [9]. Execution statistics from the simulation are further fed to a retargetable, McPAT-based reference power model of the chosen processor [16]. The resulting timing and energy estimates 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 back annotation process at the core of our flow. As illustrated in Figure 1, the back annotator consists of three main steps: building the mapping table, extracting and characterizing basic blocks and annotating characterizations into the compileable IR.

2.1 Mapping

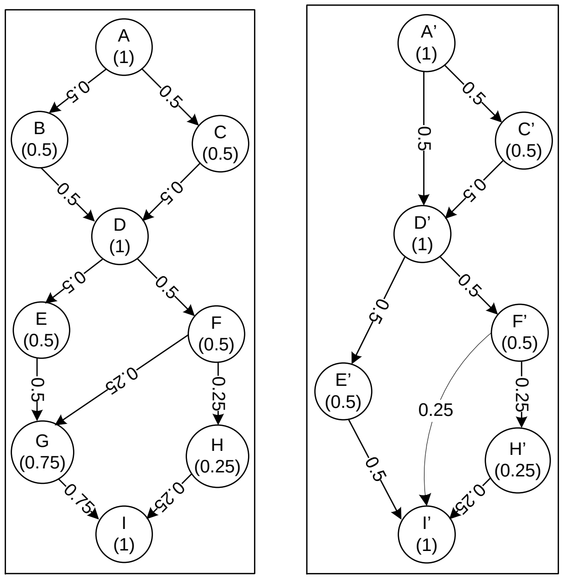
The first step in constructing a binary-IR mapping table is to build the CFGs of the compileable IR and the binary generated from it. In the IR, basic blocks are delineated by identifying their starting and ending line numbers in the code. Building the CFG for the binary follows a similar process, but it requires identifying all assembly instructions that can cause control flow changes. This invariably introduces a dependence on the target instruction set in the annotation flow. This can be mitigated, however, by automatically extracting the knowledge of control flow change instructions from the ADL description of the processor, which we plan to address in future work. Currently, we employ a separate template for pattern matching and recognition of control flow instructions.

After constructing the CFGs, a match of both graphs has to be established. Due to optimizations in the compiler backend, however, changes in the basic block structure of the code can occur and graphs will not match exactly. Typical mismatches between IR and binary CFGs are illustrated in Figure 2. In general, all changes can be reduced to nodes being added or removed on either side of the graph (Figure 2(a)) when instructions are percolated through the control flow hierarchy by the compiler. Instructions can be pushed down into successors, which can lead to additional blocks being introduced on one or more binary branch paths (Figure 2(b)). Conversely, blocks will disappear from the binary if all their instructions are determined to be path independent and subsequently are moved up into each of the block's predecessors (Figure 2(a) and (c)). This includes cases where complete if-then-else structures are replaced with predicated execution using conditional instructions in straight-line code (Figure 2(d)).

A valid graph mapping needs to be established in the presence of such variations. The gcc suite debugger, GDB, was an early candidate for extracting a mapping of binary addresses to IR code lines. Since there is generally a one-to-many mapping between blocks under optimizations, however, GDB's mapping information alone is not sufficient. We therefore use a heuristic subgraph matching algorithm that only falls back on debug information when multiple equally likely matches are possible (e.g. as is the case when trying to match the two branches of simple if-then-else structures, which can not be identified from the graph structure alone).

We know that unique mappings will exist and will have to adhere to the overall control flow of the application. As such, we perform a synchronized depth-first traversal of both CFGs and identify legal matches between pairs of blocks. 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 identify legal matches based on a control flow representation using both loop and branch nesting levels.

For the latter, we introduce a *flow* value associated with each basic block that describes the branching structure of the CFGs, as shown in Figure 3 (extracted from the ADPCM benchmark used in our experiments). The flow value of a basic block is equal to the sum of all its incoming flows, which in turn are equally divided among all outgoing edges. Root nodes of a CFG have a flow value of 1. For mapping



(a) IR CFG (b) Binary CFG

Figure 3: Flow value computation example.

Algorithm 1 Flow value computation.

```
1: function ComputeFlow
2:     $Flow(EntryBB) = 1$
3:     $Put(Succ(EntryBB), Queue)$
4:     while $|Queue| \neq 0$ do
5:         $BB = Get(Queue)$
6:         for all $S_i \in Succ(BB)$ do
7:             if $S_i \in Ancestors(BB)$ then
8:                 $DeleteEdge(S_i \leftarrow BB)$
9:             end if
10:        end for
11:        for all $P_i \in Pred(BB)$ do
12:            $Flow(BB) += \frac{Flow(P_i)}{|Succ(P_i)|}$
13:        end for
14:        $Put(Succ(BB), Queue)$
15:    end while
16: end function
```

purposes, only nodes with equal flow numbers in both graphs will be considered as potential matches. For example, in Figure 3, node G in the IR has a flow value of 0.75, which does not match any block in the binary. Algorithm 1 shows the breadth-first traversal we use to calculate the flow value across an entire CFG. Note that loop nesting levels for each node are computed separately. As such, backward edges are considered in the context of loop nests and ignored for the purpose of flow value computation.

After recording flow and loop nesting level information, we apply these as constraints for establishing candidate matches between pairs of basic blocks during actual binary-to-IR mapping (Algorithm 2). Starting from the roots of each function, basic blocks are matched by recursively evaluating all possible successor pairings and recording the one(s) with the minimum cost. Each potential match is thereby associated with a cost that is the sum of unmatched blocks in the two subgraphs rooted at each block. A pair of basic blocks will be compatible only if both flow and loop nest levels are identical. Non-comparable pairs will be associated with an infinite matching cost. The algorithm uses and returns a dictionary that records all compatible least-cost matches found between any pair of blocks.

Given a pair of basic blocks to map, the algorithm first checks for compatibility and whether the pair has already been matched. In the latter case, the previously recorded cost is returned. During the comparison of a pair of blocks, the algorithm then enumerates and evaluates the cost of all possible mappings between successors of each block. This includes cases where a successor node is skipped and remains unmatched. In each iteration, the algorithm picks successor subsets S to skip out of the powerset $\mathcal{P}(Succ(BB))$, i.e. out of the set of all possible subsets of successors (which includes the empty set). Blocks in S are then subsumed by pulling up their successors. A new successor set SS is constructed in which each chosen successor block is replaced with its own successor set. A *LocalCost* thereby represents the total number of skipped blocks in either of the two sets. If the cardinality of resulting successor sets is the same, the blocks contained in those sets are considered for further matching. All possible permutations of one-to-one correspondences m between the two sets are evaluated. For each successor mapping m, its cost is recursively computed as the sum of the *LocalCost* and the costs of all successor pair matches. Finally,

Algorithm 2 Binary-to-IR mapping.

```
1: function Mapping($BB_1$, $BB_2$)
2:     if $Flow(BB_1) \neq Flow(BB_2)$
3:  or $NestingLevel(BB_1) \neq NestingLevel(BB_2)$ then
4:         return $\infty$
5:     end if
6:     if $(BB_1, BB_2) \in MatchingDict$ then
7:         return $MatchingDict(BB_1, BB_2)$
8:     end if
9:     $Mincost = \infty$
10:    $S_1 = Succ(BB_1)$
11:    $S_2 = Succ(BB_2)$
12:    for all $S_i \in \mathcal{P}(S_1)$ do
13:        for all $S_j \in \mathcal{P}(S_2)$ do
14:            $LocalCost = |S_i \cup S_j|$
15:            $SS_1 = (S_1 - S_i) \cup \bigcup_{s \in S_i} Succ(s)$
16:            $SS_2 = (S_2 - S_j) \cup \bigcup_{s \in S_j} Succ(s)$
17:            if $|SS_1| == |SS_2|$ then
18:                for all bijections $m_k : SS_1 \rightarrow SS_2$ do
19:                    $Cost = 0$
20:                    for all $BB_l \in SS_1$ do
21:                        $Cost+ =$
22:                        Mapping($BB_l, m_k(BB_l)$)
23:                    end for
24:                    $Mincost =$
25:                    $min(Mincost, Cost + LocalCost)$
26:                end for
27:            end if
28:        end for
29:    end for
30:    $MatchingDict(BB_1, BB_2) \leftarrow Mincost$
31:    return $Mincost$
32: end function
```

the minimal matching cost among all the possible mappings of all possible successor skips will be recorded as the matching cost of the current basic block pair.

With the use of a dictionary to maintain already matched block pairs, no pair is visited more than once. Hence, algorithm complexity is *O(MN)* in the number of nodes M and N in each graph. The final mapping of binary to IR blocks is established by the least cost match recorded for each binary block in the result dictionary. Note that the dictionary may contain more than one match with equal minimum cost. In these cases, a post-processing that uses debugging information to resolve ambiguities is applied. Addresses of instructions in the binary block are successively queried for source line information until a match with a candidate IR block is found. In our experiments, our mapping heuristic returned 100% accurate matches for all test cases.

2.2 Basic Block Characterization

The second step in the back annotation process is the characterization of block-specific target metrics. Accurate characterization is complicated by the fact that the timing and energy consumption of a basic block can be significantly affected by pipeline effects such as stalls, which depend on the state of the processor at the start of execution of the block. In other words, in a real execution flow, the processor state at the entry point of a basic block, and hence the timing and energy consumption for the whole block is determined by code that has previously executed (Figure 4).

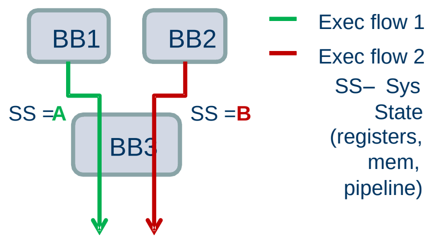


Figure 4: Path dependency of processor state.

To approximate this effect, we characterize each block through pairwise executions with all of its possible predecessors. Such a two-level approximation of the actual path history represents a tradeoff between accuracy and characterization complexity. As results will show, a two-level characterization incurs only a slight accuracy loss in a few cases.

The presence of function calls in a basic block presents additional considerations. One option is to simply remove them while characterizing the caller's basic block. The function itself would be characterized, for all its basic blocks, in isolation. However, in the same way as regular blocks, during real execution the caller and callee can influence each other's timing. We illustrate this with the help of a small example: Figure 5 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:

$$\begin{aligned} T_{Path} &= T_A + T_{B\_1|A} + T_{D|B\_1} + T_{F|D} + \\ & \quad T_{G|F} + T_{B\_2|G} + T_{C|B\_2}, \end{aligned}$$

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 Pairwise Execution

Each basic block pair is characterized by executing it on an ISS to collect cycle counts and to feed the resulting execution statistics to McPAT for energy estimation. As mentioned above, a given basic block has to be characterized as many times as the number of blocks that can precede it in all possible execution flows. Notably, for a basic block with function calls, each constituent sub-block is paired up with its predecessors and individually characterized. Sub-block timings are aggregated to get the overall timing of the block.

Before executing a basic block pair, the processor state is properly initialized to prevent exceptions due to illegal operations, such as divide by 0. The initialization is extracted from the processor state after execution of the predecessor's predecessor of the basic block being characterized. We record such state information (register and memory values) after execution of each pair. This then creates a dependency between execution of different block pairs. A scoreboarding system has been implemented to track such dependencies, where dependent executions are only enabled upon the execution of a suitable block pair.

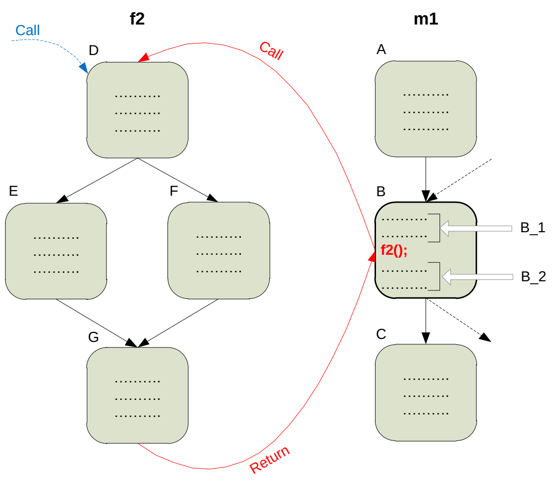


Figure 5: Function call timing.

Performing pairwise execution of basic blocks requires that conditional branches at the end of the predecessor, if any, are steered to the other block in the pair. A similar requirement exists for basic block pairs spanning a return from a function. We therefore introduce three non-architected registers into the processor's ADL description to implement steering of branch during pairwise block characterization:

Predecessor End Address (PEA) The address of the instruction for which to force branch direction, if at all.

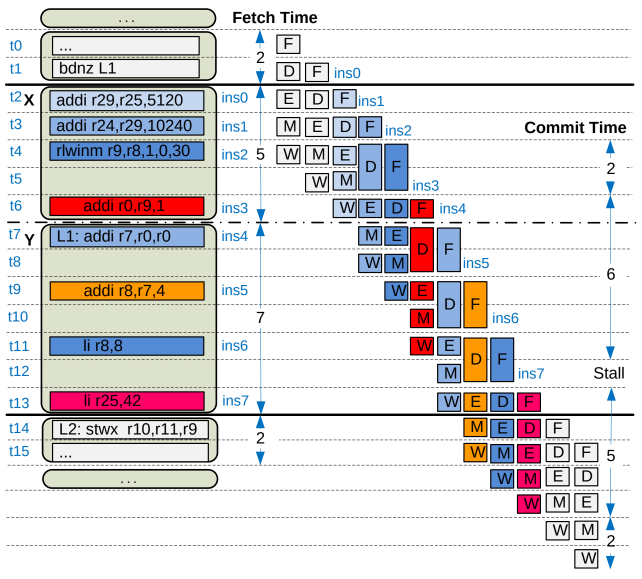
Branch Direction (BDIR) Specifies if forcing of branch directions is enabled and if so, in which direction (taken or not taken). For a particular basic block pair to force, its value is determined by comparing the branch target (if any) of the predecessor with the start address of the given successor.

Correct Return Address (CRA) If the predecessor in a basic block pair ends with a function return, then this register points to the correct instruction to return to.

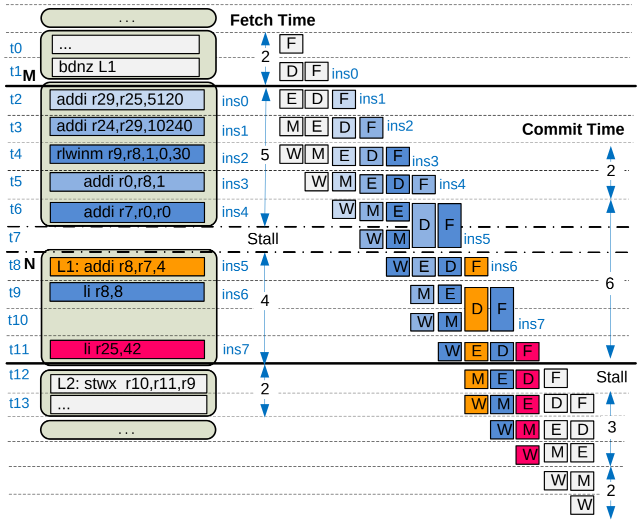
The ADL description of conditional branch and return instructions is modified so that on encountering the PEA-specified instruction, BDIR and CRA are used to guide the PC to the address it should point to next.

2.2.2 Timing Characterization

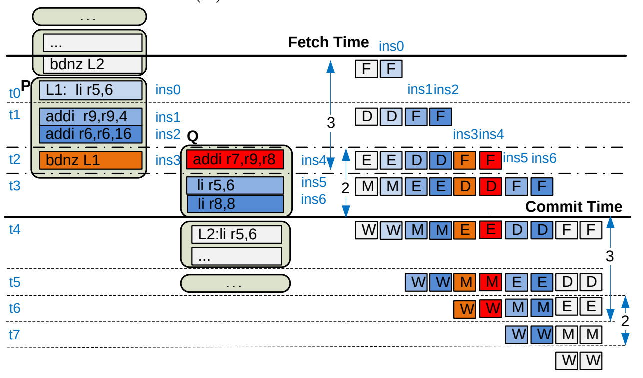
To calculate the execution time of a basic block for a certain predecessor, the detailed trace generated from its pairwise ISS execution is analyzed (Figure 6). We rely on the difference in the fetch time instants of the first and last instructions in a characterized basic block to determine its execution 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



(a) Normal execution



(b) Stall between blocks



(c) Overlap between blocks

Figure 6: Inter-block timing characterization.

fetch time of a following instruction. For example, as shown in Figure 6(a), the stalls in the execution stages and hence commit times for instructions 3, 4 and 5 are propagated to the fetching stages of instruction 4, 5 and 6, respectively. As such, they will be included in the execution time of 7 cycles annotated to basic block Y under predecessor X. In the process, both intra- and inter-block dependencies are accurately accounted for. When accumulating execution delays during simulation, this setup captures all delays associated with instructions in blocks X and Y, including the single stall between commits of instructions from both blocks.

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. This effect is illustrated in the example of Figure 6(b). The end of basic block X from Figure 6(a) has moved from instruction 3 to instruction 4. As a result, the fetching stall in instruction 5 will appear between the new basic block pair (M, N). In this case, 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 micro-architectures a situation may occur where fetch times of successive blocks start overlapping. Figure 6(c) shows an example of such a case. The pair (P, Q) overlaps on the last instruction of P. This necessitates a reduction in Q's characterized time by the amount of the overlap. In case of overlapping fetch times between the first instruction of a block and the last instruction of its predecessor, the amount of overlap is therefore subtracted from the execution time of the characterized block.

Overall, intra- and inter-block pipeline effects are accurately accounted for. Note that our pairwise characterization scheme is also able to accurately handle effects of static branch predictors. In static predictors, either the branch target or the fall-through block will always suffer a misprediction penalty at the beginning of its execution. This is similar to other basic blocks suffering a stall in their first instructions. Dynamic branch predictors are not accurately characterized using our approach. To generally capture dynamic effects, our back-annotated models can be augmented with dynamic cache and branch predictor simulation models from literature. Such extensions with existing work are, however, outside of the scope of this paper.

Table 1: Energy characterization statistics.

Overall Statistics	total cycles
Instruction Statistics	total instructions
	integer instructions
	branch instructions
	branch mispredictions
	load instructions
	store instructions
	committed instructions
	committed integer instructions
	function calls
	context switches
Operation Statistics	instruction window reads
	instruction window writes
	integer register file writes
	integer register file reads
	integer ALU accesses
	multiplier accesses
	memory accesses
	memory reads
	memory writes

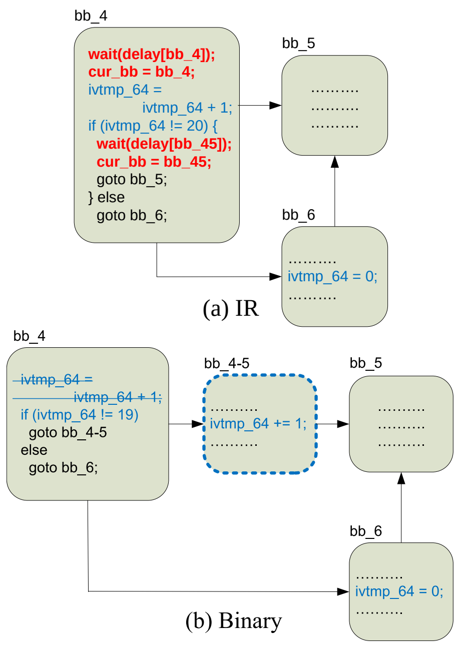


Figure 7: Annotation of bridging blocks.

2.2.3 *Energy Characterization*

McPAT, the power estimation tool used in our flow, requires execution statistics as shown in Table 1. These statistics are extracted from the detailed trace emitted during the ISS execution. When characterizing a selected block with any of its predecessor, corresponding instruction and operation statistics are collected only for instructions contained in the characterized block itself. 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.

Compared to timing, power modeling poses additional challenges. Switching activity and hence power consumption of a code block will generally be data (input) dependent. We currently annotate a single energy consumption figure (based on default McPAT activity factors) per pair of basic blocks, leading to small residual errors. In future work, we plan to develop an input-dependent characterization of the per-block energy consumption based on the actual data flowing through the block during simulated execution.

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 annotations into the compileable IR 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. Additional conditional code is inserted for tracking of sub-blocks. A logical representation of sample annotated code is shown in Figure 1.

Unmatched blocks remaining in the mapping table after applying the algorithm from Section 2.1 may require special considerations. Accounting for blocks missing in the binary is as simple as not annotating the unmatched IR block. However, 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. Figure 7 illustrates this adjustment to the annotation process. In this case, basic block *bb_4-5*, which exists in the binary, has no match in the intermediate representation. Its timing and energy information therefore has to be back-annotated onto the *bb_4* to *bb_5* edge in the IR. This is achieved by inserting corresponding annotations directly into the associated control flow statement of the source block *bb_4*.

Table 2: Benchmark summary.

Benchmark	Suite	Mods.	BBs	Sim. instr.
SHA (Sm)	Security	Disabled tail-call	50	14,674,926
SHA (Lg)				153,067,895
ADPCM (Sm)	Telecom	-	40	36,658,548
ADPCM (Lg)				724,729,999
CRC32 (Sm)	Telecom	Disable inlining	7	13,688,752
CRC32 (Lg)				266,112,122
Sieve	-	-	17	12,284,657

3. EXPERIMENTS AND RESULTS

We have implemented our automated back-annotation flow in Python using the uADL ISS [9] and McPAT [16] as timing and energy references, respectively. Our retargetable back-annotation (RBA) tool is available for download at [23].

To demonstrate the automation of our flow for host compiled model development, we applied it to several standard benchmarks running on two generic PowerPC based targets. Back annotations and simulations were performed on a quad-core Intel i7 workstation running at 2.6 GHz.

We selected three benchmarks from the MiBench suite with both small and large data sets [18] as well as a custom application (Eratosthenes' Sieve) for validating our flow. The latter calculates prime numbers in the range 0 to 500,000. Since back-annotation requires source code to be available, in the presence of library calls, we included library code in the characterization. If library sources are not available, e.g. in case of floating-point emulation on our PowerPC targets, back-annotation needs to resort to static or statistical estimation of library timing [31]. Since this can lead to additional inaccuracies on top of evaluating the base accuracy of our flow, we excluded such benchmarks in this paper. Benchmarks were further modified to validate proper function call characterization.

Table 2 shows a summary of the benchmarks and the modification made to the original code or compilation process. All benchmarks were changed to accept constant array versions of the inputs that replace the file I/O in the original code. Benchmarks were cross-compiled to the IR and binary levels using powerpc-elf-gcc with the 'O2' optimization level. C conversion and back-annotation of the IR code was performed using our automated scripts. Finally, resulting host-compiled simulation models were executed under both small and large input sets provided in the MiBench suite.

The first target we evaluated is an in-order, e200_z4-like dual-issue core with no cache, MMU, floating point unit or dynamic branch predictor. The other target is an e200_z6-like single-issue core with a longer pipeline. A gcc-4.4.5 cross-compiler was used to generate code for both targets.

Table 3: Runtime Comparison

	z4-like core						z6-like core					
	Host-Compiled			ISS			Host-Compiled			ISS		
Benchm.	BA	Sim.	Total	Sim.	McPAT	Total	BA	Sim.	Total	Sim.	McPAT	Total
SHA (Sm)	02:42.7	9ms	02:42.7	00:15.2	1.72s	00:17.0	02:49.2	11ms	02:49.2	00:17.5	1.67s	00:19.2
SHA (Lg)		44ms	02:42.7	02:37.0	1.77s	02:38.8		44ms	02:49.2	03:04.9	1.79s	03:06.6
CRC (Sm)	00:31.2	9ms	00:31.2	00:17.2	1.69s	00:18.8	00:32.0	11ms	00:32.0	00:18.6	1.67s	00:20.3
CRC (Lg)		87ms	00:31.2	06:27.1	1.73s	06:28.8		86ms	00:32.1	06:10.3	1.79s	06:12.0
ADP. (Sm)	02:04.9	24ms	02:04.9	00:38.3	1.82s	00:40.1	02:02.1	24ms	02:02.1	00:42.9	1.67s	00:44.5
ADP. (Lg)		317ms	02:05.2	12:27.4	1.84s	12:29.2		323ms	02:02.4	13:29.1	1.75s	13:30.9
Sieve	00:47.0	13ms	00:47.0	00:14.9	1.67s	00:16.5	00:47.3	12ms	00:47.4	00:15.3	1.70s	00:17.0

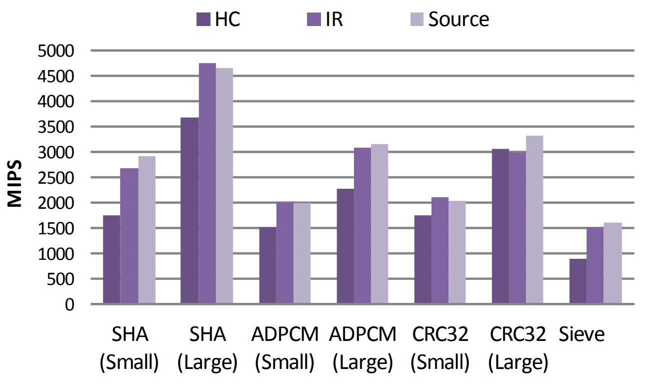


Figure 8: Simulation speed.

3.1 Speed Results

Table 3 shows runtimes of the back-annotation (BA) process for generation of final host-compiled simulation models from the original C source code, which includes cross-compilation, IR conversion and timing and energy characterization. Furthermore, simulation times of host-compiled models are compared against ISS and McPAT runtimes in a traditional simulation setup. As results show, for the smallest data sets and short simulation runs, 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. Furthermore, back-annotation is a one-time effort. Once generated, resulting host-compiled models can be repeatedly resimulated under varying input scenarios.

Figure 8 shows the simulation speeds of the host-compiled (HC) models as compared to that of source and IR level simulations. By contrast, the ISS runs at about 0.8-1.0 MIPS. As is evident, there is no degradation in simulation speed by working with the IR instead of the source. On an average, the host-compiled models are nearly 2100 times faster than the ISS, while being marginally slower than the IR.

3.2 Timing and Energy Results

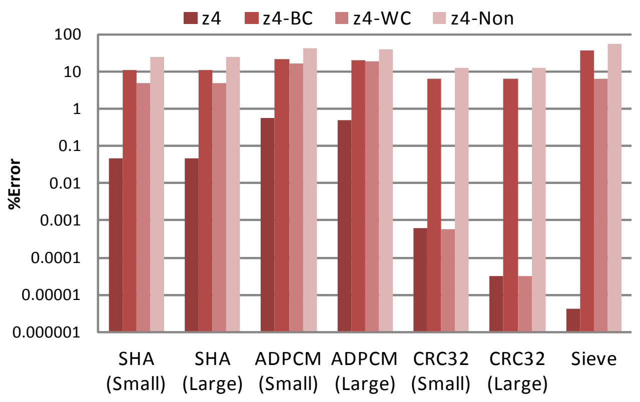
Figure 9 and 10 compare the accuracy of timing and energy results obtained from the host-compiled (HC) models against cycle-accurate simulations of complete application runs executed on the respective ISS+McPAT reference models. To verify that our pairwise characterization approach indeed adds significant accuracy, we replicated modified experiments aimed at comparing our results against existing approaches in literature that rely on static timing or energy characterizations only. We ran experiments for both z4 and z6 cores in which each block was annotated with a single value only. In a first setup, basic blocks were characterized in isolation without any predecessor dependencies being considered at all (Non). Other experiments use the best-case (BC) or worst-case (WC) timing and energy values obtained over all predecessors. The latter resemble classical approaches that are based on static best-case and worst-case characterization. Note, however, that our simulation-based approach is likely to still remain less conservative and hence more accurate than true best and worst analysis.

Resulting timing and energy errors are summarized in Tables 4 through 7. 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, our flow shows maximum timing and energy errors of 2.2% and 0.8% with an average error of 0.8% and 0.4%, respectively. Residual errors, especially for the z6-ADPCM combination, are due to dependencies spanning more than two basic blocks, which are not captured by our flow. Static characterizations following traditional approaches lead to energy errors between 10% and 20%, and timing errors ranging from 20% up to 50%. Overall, results confirm that pairwise characterization represents a good tradeoff between significantly improved accuracy and low back-annotation runtime.

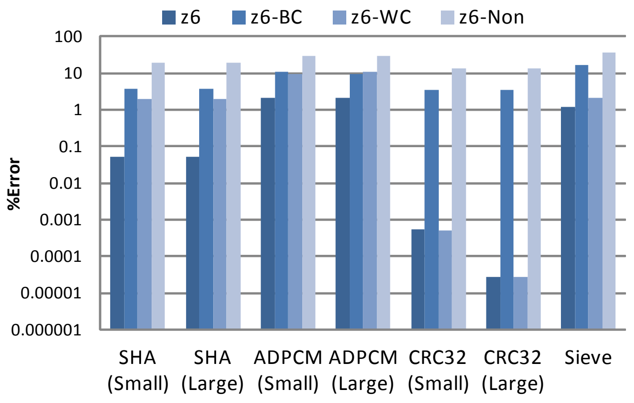
4. SUMMARY AND CONCLUSIONS

In this paper, we presented a framework for combined, fast and accurate power (energy) and performance estimation based on 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 power 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 simulation for the host-compiled approach as compared to a cycle accurate ISS. Utilizing path-dependent, pairwise and simulation-based characterization at the basic block level, accuracy is significantly improved compared to approaches that annotate a single metric per block. Overall, simulation speed remains close to native execution at near cycle accuracy.

Future work will be concerned with further automation of the approach, including 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.

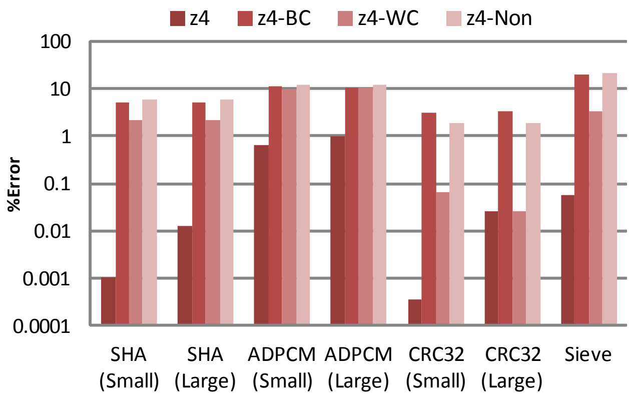


(a) z4-like core

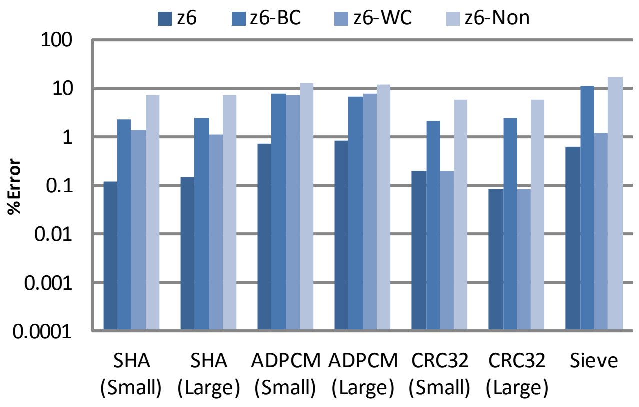


(b) z6-like core

Figure 9: Host-compiled timing accuracy.



(a) z4-like core



(b) z6-like core

Figure 10: Host-compiled energy accuracy.

5. ACKNOWLEDGMENTS

This work has been supported by SRC award 2085. We also would like to thank Gary Morrison for his invaluable help throughout the project.

6. REFERENCES

[1] A. Bouchhima, P. Gerin, and F. Petrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *ASP-DAC*, 2009.
[2] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimates based on statistical characterisation of intermediate compilation code. In *ISLPED*, 2011.
[3] C. Brandolese et al. A multi-level strategy for software power estimation. In *ISSS*, 2000.
[4] C. Brandolese et al. Source-level execution time estimation of C programs. In *CODES*, 2001.
[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.
[6] L. Cai, A. Gerstlauer, and D. Gajski. Multi-metric and multi-entity characterization of applications for early system design exploration. In *ASPDAC*, 2005.
[7] D. Calvo et al. A multi-processing systems-on-chip native simulation framework for power and thermal-aware design. *Journal of Low Power Electronics*, 7(1):2–16, 2011.
[8] E. Copty, G. Kamhi, and S. Novakovsky. Transaction level statistical analysis for efficient microarchitecture power and performance studies. In *DAC*, 2011.
[9] Freescale. ADL Release 2.0.0. `http://opensource.freescale.com/fsl-oss-projects`.
[10] A. Gerstlauer et al. Abstract system-level models for early performance and power exploration. In *ASP-DAC*, 2012.
[11] A. Goswami and A. Gerstlauer. ExtractCFG: A framework to enable accurate timing back annotation of c language source code. Technical Report UT-CERC-11-02, CERC, UT Austin, Aug. 2011.
[12] F. Herrera et al. A MDD methodology for specification of embedded systems and automatic generation of fast configurable and executable performance models. In *CODES+ISSS*, 2012.
[13] C. Hu, D. A. Jiménez, and U. Kremer. Efficient program power behavior characterization. In *HIPEAC*, 2007.
[14] E. Y. Hwang, S. Abdi, and D. Gajski. Cycle approximate retargettable performance estimation at the transaction level. In *DATE*, 2008.
[15] M. Krause et al. Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation. In *CODES+ISSS*, 2008.
[16] S. Li et al. McPAT: An integrated power, area, and timing modeling framework for multicore and

Table 4: Timing results and errors for the z4-like core.

Benchmark	ISS [cycles]	HC [cycles]	HC-BC [cycles]	HC-WC [cycles]	HC-Non [cycles]
SHA (Sm)	21,681,024	21,670,904 (<0.1%)	19,248,234 (11.2%)	22,732,162 (4.8%)	16,455,492 (24.1%)
SHA (Lg)	226,146,935	226,042,525 (<0.1%)	200,765,433 (11.2%)	237,111,886 (4.8%)	171,640,521 (24.1%)
ADPCM (Sm)	68,100,669	67,716,284 (0.6%)	53,155,344 (21.9%)	79,159,420 (16.2%)	40,005,067 (41.3%)
ADPCM (Lg)	1,319,358,706	1,312,764,801 (0.5%)	1,056,918,002 (19.9%)	1,562,535,662 (18.4%)	787,123,388 (40.3%)
CRC32 (Sm)	21,901,999	21,901,865 (<0.1%)	20,533,002 (6.3%)	21,901,868 (<0.1%)	19,164,132 (12.5%)
CRC32 (Lg)	425,779,393	425,779,257 (<0.1%)	399,168,057 (6.3%)	425,779,260 (<0.1%)	372,556,850 (12.5%)
Sieve	23,277,599	23,277,598 (<0.1%)	14,696,677 (36.9%)	24,735,520 (6.3%)	10,167,396 (56.3%)

Table 5: Energy results and errors for the z4-like core.

Benchmark	McPAT	HC	HC-BC	HC-WC	HC-Non
SHA (Sm)	11.7 mJ	11.7 mJ (<0.1%)	11.1 mJ (5.2%)	12.0 mJ (2.2%)	11.1 mJ (5.7%)
SHA (Lg)	122.6 mJ	122.6 mJ (<0.1%)	116.3 mJ (5.1%)	125.4 mJ (2.3%)	115.6 mJ (5.7%)
ADPCM (Sm)	32.8 mJ	32.6 mJ (0.6%)	29.0 mJ (11.6%)	36.0 mJ (9.8%)	28.8 mJ (12.2%)
ADPCM (Lg)	654.9 mJ	639.5 mJ (1.0%)	576.3 mJ (10.8%)	712.8 mJ (10.4%)	570.3 mJ (11.7%)
CRC32 (Sm)	11.1 mJ	11.1 mJ (<0.1%)	10.8 mJ (3.2%)	11.1 mJ (<0.1%)	10.9 mJ (1.9%)
CRC32 (Lg)	216.1 mJ	216.2 mJ (<0.1%)	209.1 mJ (3.2%)	216.2 mJ (<0.1%)	212.0 mJ (1.9%)
Sieve	10.7 mJ	10.7 mJ (<0.1%)	8.6 mJ (19.5%)	11.1 mJ (3.4%)	8.4 mJ (21.3%)

Table 6: Timing results and errors for the z6-like core.

Benchmark	ISS [cycles]	HC [cycles]	HC-BC [cycles]	HC-WC [cycles]	HC-Non [cycles]
SHA (Sm)	37,212,392	37,192,461 (0.1%)	35,776,842 (3.9%)	37,966,641 (2.0%)	30,278,570 (18.6%)
SHA (Lg)	388,147,772	387,941,772 (0.1%)	373,179,244 (3.9%)	396,016,851 (2.0%)	315,824,490 (18.6%)
ADPCM (Sm)	110,480,693	112,777,478 (2.1%)	98,625,819 (10.7%)	121,228,058 (9.7%)	77,353,333 (30.0%)
ADPCM (Lg)	2,153,169,608	2,200,081,262 (2.2%)	1,946,171,541 (9.6%)	2,385,256,740 (10.8%)	1,516,457,184 (29.6%)
CRC32 (Sm)	39,697,348	39,697,135 (<0.1%)	38,328,272 (3.4%)	39,697,138 (0.0%)	34,221,662 (13.8%)
CRC32 (Lg)	771,725,124	771,724,908 (<0.1%)	745,113,708 (3.4%)	771,724,911 (0.0%)	665,280,087 (13.8%)
Sieve	35,106,417	35,523,127 (1.2%)	29,048,242 (17.3%)	35,856,064 (2.1%)	22,402,866 (36.2%)

Table 7: Energy results and errors for the z6-like core.

Benchmark	McPAT	HC	HC-BC	HC-WC	HC-Non
SHA (Sm)	15.5 mJ	15.5 mJ (0.1%)	15.1 mJ (2.2%)	15.7 mJ (1.4%)	14.4 mJ (6.8%)
SHA (Lg)	162.1 mJ	161.9 mJ (0.1%)	158.2 mJ (2.4%)	163.9 mJ (1.1%)	150.5 mJ (7.1%)
ADPCM (Sm)	43.2 mJ	43.5 mJ (0.7%)	40.0 mJ (7.4%)	46.2 mJ (7.0%)	37.8 mJ (12.5%)
ADPCM (Lg)	847.6 mJ	854.4 mJ (0.8%)	791.8 mJ (6.6%)	912.3 mJ (7.6%)	746.5 mJ (11.9%)
CRC32 (Sm)	15.4 mJ	15.4 mJ (0.2%)	15.1 mJ (2.1%)	15.4 mJ (0.2%)	14.5 mJ (5.5%)
CRC32 (Lg)	300.2 mJ	300.0 mJ (<0.1%)	293.2 mJ (2.3%)	300.0 mJ (<0.1%)	282.8 mJ (5.8%)
Sieve	13.6 mJ	13.7 mJ (0.6%)	12.1 mJ (11.2%)	13.8 mJ (1.2%)	11.4 mJ (16.3%)

manycore architectures. In *MICRO*, 2009.

[17] 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.

[18] MiBench Version 1.0. `http://www.eecs.umich.edu/mibench/`.

[19] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann. Control-flow-driven source level timing annotation for embedded software models on transaction level. In *DSD*, 2011.

[20] L. Murillo et al. Synchronization for hybrid MPSoC full-system simulation. In *DAC*, 2012.

[21] Y.-H. Park et al. A multi-granularity power modeling methodology for embedded processors. *IEEE TVLSI*, 19(4):668–681, 2011.

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

[23] RBA Version 1.0. `http://www.ece.utexas.edu/~gerstl/releases/`.

[24] G. Schirner, A. Gerstlauer, and R. Dömer. Fast and accurate processor models for efficient MPSoC design. *ACM TODAES*, 15(2):1–26, Mar. 2010.

[25] J. Schnerr et al. High performance timing simulation of embedded software. In *DAC*, 2008.

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

[27] D. Sunwoo, G. Wu, N. Patil, and D. Chiou. PrEsto: An FPGA-accelerated power estimation methodology for complex systems. In *FPL*, 2010.

[28] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE TVLSI*, 2:437–445, Dec. 1994.

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

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

[31] Z. Wang, A. Sanchez, and A. Herkersdorf. Fast and accurate software performance estimation during high-level embedded system design. In *WOSP*, 2008.

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