Source-Level Performance, Energy, Reliability, Power and Thermal (PERPT) Simulation

Zhuoran Zhao, *Student Member, IEEE,* Andreas Gerstlauer, *Senior Member, IEEE,* Lizy K. John, *Fellow, IEEE*

***Abstract*—With ever increasing design complexities, traditional cycle-accurate or instruction-set simulations are often too slow or too inaccurate for system prototyping in early design stages. As an alternative, host-compiled or source-level software simulation has been proposed, but existing approaches have largely focused on timing simulation only. In this paper, we propose a novel source-level simulation infrastructure that provides a full range of performance, energy, reliability, power and thermal (PERPT) estimation. Using a fully automated, retargetable back-annotation framework, intermediate representation code is statically annotated with timing, energy and resource accesses information obtained from low-level references at basic block granularity. The annotated model is natively compiled and combined with a cache model and occupancy analyzer to provide target performance, energy, soft-error vulnerability and power estimations. Finally, generated power traces are fed into thermal models for further temperature estimation.**

Comprehensive evaluations of our source-level models for PERPT estimations are performed. We applied our approach to PowerPC targets running various industry benchmark suites. Source-level simulations are evaluated for different PERPT metrics and with cache models at various levels of detail to explore the speed and accuracy tradeoffs. More than 90% accuracy can be achieved for timing, energy, reliability and power estimation, and an average error of 0.05K exists in steady-state thermal estimation. Simulation speeds range from 180 MIPS to 5740 MIPS for different types of metrics at different abstraction levels.

***Index Terms*—Host-compiled and source-level simulation, virtual platform prototyping.**

I. Introduction

Embedded systems are usually designed under tight design schedules and budgets. In early design stages, architectures and applications have to be co-designed across a large hardware/software space with multiple simultaneous optimization objectives. Various implementation choices and multi-dimensional evaluation metrics make the design space extremely large, and fast yet accurate evaluation methodologies are required.

Simulations usually play a crucial role in the validation process due to their ability to accurately capture the dynamic behavior and interactions across the system stack. Designers typically rely on executable models for accurate feedback on various metrics of their candidate designs. Traditionally, instruction set simulators (ISSs), cycle-accurate, micro-architectural or RTL/gate-level descriptions have been used to perform simulations of software applications executing on a target platform. Their drawback is that they are either inaccurate or slow, since they require the processor micro-architecture to be either fully abstracted or modeled in detail.

Copyright (c) 2015 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

High-level software and processor models based on native, so-called host-compiled or source-level software execution have recently emerged as fast and accurate alternatives [1]. 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. Execution statistics are 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 source-level approaches thus far have mostly focused on timing estimation only. However, continuously increasing integration densities and physical limits of technology scaling have made power, thermal and reliability concerns crucial design metrics that also need to be considered. In this paper, we propose a novel source-level software modeling and simulation infrastructure that encompasses a full range of performance, energy, reliability, power and thermal (PERPT) metrics. Our flow is fully automated and retargetable. It is built by annotating the compiler generated intermediate representation (IR) of the application source code with estimates obtained from reference models. The annotated model is then executed natively on a host machine to generate PERPT estimations. In previous work, we developed basic timing and energy estimation [2]. In this paper, we extend our prior work by considering additional micro-architecture effects, such as dynamic cache behavior, while also providing a wider range of PERPT metrics. To provide soft error vulnerability estimations, code is back-annotated with information to trace accesses to micro-architectural structures, and an online occupancy analyzer is run along with the simulation. Furthermore, power traces for each floorplan component are recorded periodically at a predefined sampling rate. Finally, generated power traces are forwarded to thermal models for temperature estimation and hotspot identification [3].

Comparing with previous work, the specific contributions of this paper are: (1) an extension of source-level timing, energy and thermal models to incorporate cache and memory effects; (2) a novel back-annotation based approach for fast and accurate source-level reliability estimation of register file and data cache vulnerabilities against soft errors; (3) an extension of the back-annotation infrastructure for generating transient power traces directly at the source level; and (4) a comprehensive evaluation of source-level software models across all PERPT metrics to explore speed and accuracy tradeoffs.

The rest of the paper is organized as follows: After an overview of related work and the back-annotation flow in the following sections, techniques of back annotation and host-

compiled simulation are discussed in Sections IV, V and VI. Section VII then discusses the results of our experiments and Section VIII presents the conclusions.

II. Related Work

For timing estimation, functional or cycle-accurate ISS models are widely used as a key component in virtual prototyping platforms [4], [5], [6]. Such models either rely on slow, detailed cycle-accurate micro-architecture simulation, or provide inaccurate or no timing feedback when using purely functional binary translation.

Many advanced simulation techniques have been introduced as alternatives to such execution-driven approaches. Trace-driven approaches [7] provide a flexible and scalable simulation infrastructure by separating functional from timing simulations and reducing overhead when co-simulating multiple system components. A trace-driven multi-core cache simulator is proposed in [8]. By employing PIN as a trace generator, it can characterize single-/multi-threaded applications at a speed of 4-10 MIPS with 4% error. Sniper is another PIN-based simulator that combines trace-driven with interval simulation [9]. Compared with real hardware, it can achieve 89% accuracy at speeds ranging from 450 KIPS to 5.5 MIPS under various configurations for different speed-accuracy tradeoffs. In [10], a trace-driven infrastructure for simulating parallel architectures running multi-threaded applications is shown. Instead of showing quantitative speed benefits, the paper emphasizes its scalability from 8-128 host cores. [11] demonstrates a framework for timing analysis of MPSoC architectures using abstract-timed traces, providing a 4x speedup with 95% accuracy compared to a cycle-accurate full-system model. In [12], a trace-driven MPSoC simulator with reduced synchronization overhead is proposed, achieving up to 11x speedup with 5% estimation error compared to a commercial SystemC co-simulation framework. In all cases, however, timing-accurate simulation of software on a single core as targeted in this work still requires large execution traces to be pre-generated and replayed on detailed and slow micro-architecture models. By contrast, source-level approaches statically derive a fast, coarse-grain timing model that is driven by a high-level functional simulation. Such source-level component models can in turn be used as alternative trace generators in a trace-driven system simulation framework.

Previous source-level works typically rely on complex analysis using IR-level and debug information to establish a mapping between the target binary and source code for timing back-annotation [13], [14], [15], [16]. In our work, we back-annotate IR code directly, which simplifies the mapping. Furthermore, in the presence of optimizations, we have found debug information alone 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 is described in [17]. To further increase accuracy compared to existing approaches, we perform pairwise characterization of basic blocks across all possible predecessors using execution on a cycle-accurate reference model. Similar approaches for path-dependent timing characterization can be found in [18], [19]. Finally, in order to consider memory effects and drive a high-level cache model [20], we reconstruct target memory traces solely based on IR and debugger information. Similar works [21], [22], [23] usually require binary analysis and are not fully retargetable. Our approach is comparable in speed and timing accuracy to previous works. In [13], more than 80% accuracy can be achieved for different optimization options and model details, where throughputs of 520-2500 MIPS are reached for simulations with and without cache. In [14], a WCET analyzer is integrated for binary timing profiling. 5-400 MIPS simulation throughput is achieved with errors reaching up to 15%. In [24], results show up to 13% error for non-optimized code at 10x-1000x speedup compared to a cycle-approximate multicore reference simulator. In [19], speeds of 400-1000 MIPS at errors of less than 2% are reported. None of these approaches consider energy, reliability, power or thermal metrics, however.

For power estimation, popular approaches rely on detailed, low-level macro models for micro-architectural functional blocks [25], [26]. At higher levels, existing power estimation approaches employ coarse-grain models that assume a constant or statistical energy consumption at the granularity of complete instructions, source-level operations, program phases or processor states [27], [28], [29], [30]. The authors in [29] rely on static characterization of a target ISA to perform IR-level energy estimation with up to 13% error at 400x speedup compared to a target ISS. By contrast, we make use of existing low-level reference models that operate at detailed micro-architectural granularity to back-annotate high-level IR code. By characterizing blocks in static pairs, we are able to maintain the accuracy of such low-level models while achieving fast estimation and simulation times.

For reliability modeling, we make use of the Architectural Vulnerability Factor (AVF) concept, which was introduced as a metric for measuring architecture-level soft-error reliability [31]. AVF is defined as the probability of an error occurring in a particular architecture component resulting in explicit execution errors. By its definition, AVF can be obtained through profiling of the occupancy of so-called Architecturally Correct Execution (ACE) bits, for which any error will manifest itself as a fault at the program output. Fu *et al.* [32] characterize AVF using cycle-accurate simulations across the entire program execution. Machine learning based and analytical approaches have been proposed to deal with the simulation time problem [33], [34], [35]. Such approaches, however, fail to capture many of the dynamic complexities introduced by modern software and hardware. Different from previous work, we estimate the AVF without the need for expensive instruction set simulation by leveraging fast source-level software execution and high-level processor model abstraction. Back-annotation of statically profiled target occupancy characteristics ensures estimation accuracy, while host simulation accurately captures all dynamic application/architecture interactions.

For thermal modeling, HotSpot [36] is widely used. However, this usually requires a previously collected power trace generated by execution of applications on cycle-accurate simulators and power estimation tools [37]. Alternatively, lightweight and approximate models for thermal estimation

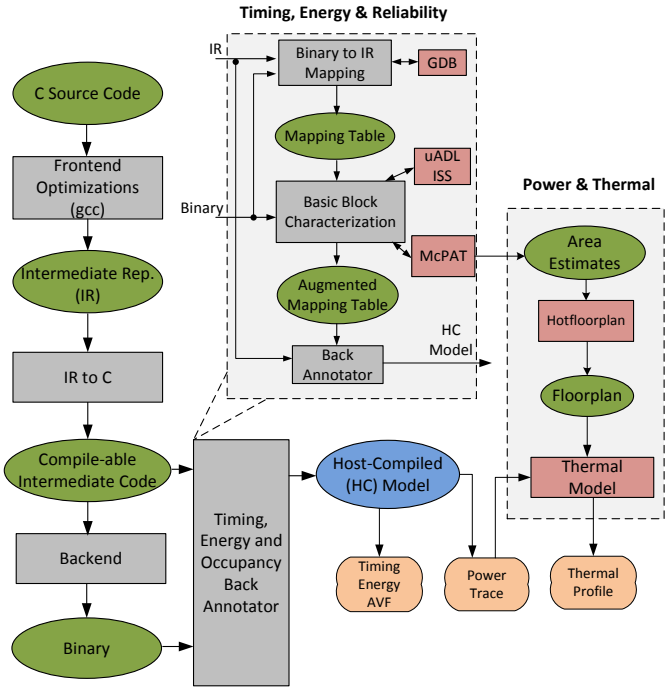


Fig. 1. Retargetable back-annotation (RBA) flow for source-level PERPT simulation.

can speed up early design space exploration significantly, albeit at the cost of accuracy [38], [39], [40]. However, these methodologies do not provide an integrated approach for thermal analysis, and the input is primarily some existing power trace. Thiele *et al.* [41] approximate the complex system of differential equations for temperature evaluation to linear equations using a Discrete-Time Temperature Evaluation Model (DTTEM) [42] for small and constant time intervals. Nevertheless, parameters for thermal analysis are again obtained using slow low-level simulation methods. In our work, we instead rely on the host-compiled source level model annotated with basic block energy consumptions to generate the power traces for each floorplan component. These power traces are then further fed into thermal models to provide quick feedback.

III. Flow Overview

Fig. 1 shows our flow for host-compiled performance, energy, reliability, power and thermal (PERPT) modeling. 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 compilable C form. Working at the IR allows typical compiler front-end optimizations to be taken into account with little or no penalty in execution speed. Moreover, the IR inherently provides a close representation of the final control flow graph (CFG) of the target code. Hence, it is able to accurately reflect all data-dependent execution behavior. In addition, 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 maps all variables and constants into a host data type of target-equivalent size and alignment. During this process, global and stack variables are extracted and corresponding target memory access traces are reconstructed based on available information from the IR and target debugger. These memory traces are then used to drive a high-level cache model that accurately emulates dynamic cache effects for estimation of cache- and memory-access related target metrics.

During following back annotation, the IR's CFG is then further augmented with timing, energy and architectural occupancy 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 the binary. This mapping is needed to accurately determine annotation points in the IR. Basic blocks (BBs) in the binary are characterized by executing them pairwise on a retargetable, cycle-accurate ISS, which is automatically generated from an open-source ADL infrastructure [5]. Execution statistics from the simulation are further fed to a retargetable reference power model [26] as well as related analysis scripts. Finally, resulting timing, energy and architectural access and occupancy information is back-annotated into the compilable IR, aided by the mapping. This creates the host-compiled model. In host simulation, timing and energy numbers for each basic block are accumulated to estimate the overall performance and energy. At the same time, dynamic register accesses and target memory traces are generated, while the occupancy information of the register files and data caches are recorded and their AVFs are further calculated.

In order to model thermal profiles, temperature estimation models, such as HotSpot and DTTEM, are incorporated into our flow. We extend back-annotation such that the energy dissipation estimates in the IR can be used to generate a structurally accurate power trace for each floorplan component. In order to model the spatial distribution of temperatures, thermal estimation tools also require the floorplan of a chip. Necessary area and placement estimates of various blocks can be obtained using McPAT and the hotfloorplan utility. Combined, the power trace and the floorplan can be utilized by the integrated thermal model to generate an online thermal profile during simulation.

Overall, our work focuses on providing a lightweight simulation infrastructure for comprehensive PERPT profiling and estimation. To the best of our knowledge, this is the first work aimed at providing a multi-metric source-level software simulation framework. The automated back-annotation and host-compiled model generation process is the core of our infrastructure. As illustrated in Fig. 1, our retargetable back-annotation (RBA) tool consists of three main steps: (1) binary-to-IR mapping, (2) basic block characterization, and (3) annotation of target models into the compilable IR. In the following subsections, we will describe these steps in more detail.

IV. Binary-to-IR Matching

The first step in our back annotation flow is to establish a mapping between target binary and compilable IR. This includes CFG matching and target memory access reconstruction, which ultimately allow target basic block metrics to be

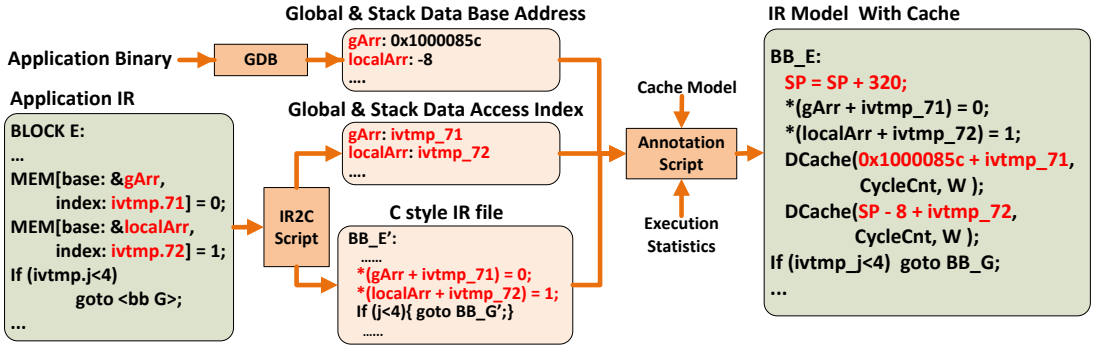


Fig. 2. Memory access trace reconstruction.

annotated back into the IR at correct insertion points while also taking cache- and memory-related effects into consideration.

A. Control Flow Graph Mapping

The first step in constructing a binary-IR mapping is to build the CFGs of the compilable IR and the binary generated from it. Using extracted CFGs, a valid graph mapping needs to be established in the presence of variations between the two graphs due to compiler backend optimizations. We 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. Debugger information is used when multiple equally likely matches are possible, as is the case in branches of if-then-else statements. Details of this mapping process can be found in [2].

B. Target Memory Trace Reconstruction

Caches can have a large effect on overall PERPT metrics. At the same time, cache behavior is highly dynamic and strongly depends on the actual sequence of memory accesses made by the application, which can not be fully determined statically during basic block characterization. We instead annotate the IR with the types and addresses of memory accesses made by the target in each basic block. Based on this information and the sequence of basic blocks traversed during program execution, memory access traces are generated during host simulation and fed into a lightweight, online cache model for estimation of related metrics. In our work, we only consider memory accesses to global and stack data. Heap accesses can be taken into account by incorporating a heap manager model that emulates the target's dynamic memory allocation and deallocation [23]. Given that most embedded applications do not employ dynamic memory, however, this is out of the scope of this paper and part of our future work.

In order to identify global or stack accesses in the IR, we parse the IR code to extract a list of all global and local variable names. For local variables, the target debugger (GDB) is invoked (using the *info address symbol* command) to further check whether they are allocated in registers or on the stack. Variables in registers are excluded from the list. We thereby assume that the IR already incorporates corresponding memory optimizations, and that local variables live exclusively on the stack or in a register. We further assume that all expressions in the IR involving accesses to global and local variable names will correspond to memory accesses in the target binary. We thus back-annotate matching memory/cache model calls at all such access points. For both global and stack accesses, the memory operation type (read or write) is identified by further analyzing these expressions. The addresses reconstruction process is described below:

1) Memory Access on Global Data: In order to reconstruct the addresses of global data accesses, their base addresses and, in case of non-scalars, their access offsets are required. A key observation is that access offsets in the IR are the same as in the target binary, while only base addresses differ. Base address information of global data can easily be obtained from the symbol table of the target ELF file by querying the debugger. By contrast, access offsets are extracted from the IR code by parsing corresponding C variables or expressions. We analyze and extract such offset information during the IR-to-C conversion stage. As shown in the example of Fig. 2, the base address of global array *gArr* is obtained from debugger as *0x1000085c*. The name of the local variable *ivtmp_71* used as index into the *gArr* array is extracted by parsing the corresponding *MEM[]* access operation in the IR. The memory access is translated into C syntax by the IR-to-C conversion script, and the base and offset information is used to annotate a matching invocation of the data cache model into the IR. For the latter, the base address of *gArr* is converted into its proper target value and combined with the dynamic value of *ivtmp_71*. With this, correct target addresses are calculated and reconstructed in the source-level model at runtime.

2) Memory Access on Stack Data: Different from the global data, stack accesses are more complicated to back annotate. Their base addresses change dynamically depending on the local and global execution context. This requires reconstructing both the target stack frame layout as well as the dynamic value of the stack pointer during program execution. Previous work relying on parsing the original C code does not clearly resolve this problem [22]. The precise allocation of stack frames is only decided after the compiler backend optimization stage. Thus, one can not rely on the application's IR or C code to deduce the stack layout. Instead, we reconstruct the base addresses of local variables using information back-annotated from the target binary. We introduce a *SP* variable to track the target stack pointer in the IR. During characterization of basic blocks, stack pointer changes are recorded and back-annotated into the IR. In this manner, the *SP* variable accurately reflects

the value of the target stack pointer at any point during host-compiled execution. Similar to global variables, we then rely on the debugger to obtain the base address *offset* of local variables relative to the current stack frame and stack pointer. In the example shown in Fig. 2, the base address offset of *localArr* is determined to be -8, the variable used to index into the array is *ivtmp_72*, and the annotated target address value becomes *SP - 8 + ivtmp_72*.

V. 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 PERPT metrics 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 beginning of a basic block, and hence the performance metrics for the whole block are determined by code that has previously executed.

To approximate this effect, we characterize each block through pairwise execution with all of its possible predecessors. Such an approximation of the actual path history represents a tradeoff between accuracy and characterization complexity. The possible extent of history effects depends on the basic block length and pipeline depth of the target processor [19]. Including additional levels of predecessors increases characterization time exponentially. As results will show, a two-level characterization incurs only a slight accuracy loss in a few cases while providing a reasonable characterization runtime.

A. Timing Characterization

To calculate the execution time of a basic block for a certain predecessor, the detailed trace generated from its pair-wise ISS execution is analyzed [2]. We rely on the difference in the fetch time instants of the first and last instructions in a characterized block to determine its execution time. This is equivalent to recording commit and hence overall execution times. In addition, we adjust for gaps triggered by stalls or multiple-issue overlaps in fetch times of successive blocks. Overall, intra- and inter-block pipeline effects are accurately accounted for. Note that our pairwise characterization is also able to accurately account for 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 handled in the same way as other basic blocks suffering a stall in their first instructions. For dynamic branch predictors, the IR code can be augmented with a simulation model of the predictor [43]. This is, however, outside of the scope of this paper.

We characterize basic blocks under the assumption that every memory access is a cache hit. Before each pairwise execution, all cache lines are initialized to be valid, and best-case execution times are back-annotated. We later account for variations in memory access delays by adding an optional penalty to the overall execution time for every back-annotated memory access. In typical embedded in-order pipelines, memory-related stalls manifest themselves as additive cycles that directly correspond to dynamically varying memory access latencies. In order to later compute such access latencies, the target cache access outcome (hit or miss) will be calculated in the online cache model, and a corresponding dynamic miss penalty will be added during source-level simulation (see Section VI).

B. Energy and Power Characterization

Energy estimation requires activity statistics to estimate the dynamic power dissipation of each component in the target processor micro-architecture. These statistics are extracted from the detailed trace emitted during ISS execution. When characterizing a selected block with any of its predecessors, corresponding instruction and operation statistics are collected only for instructions contained in the characterized block itself. Statistics are then fed into a micro-architectural energy model such as McPAT to compute predecessor-dependent power estimates for each block. Combining these estimates with the block's characterized execution timing, the power consumption obtained for each block and each floorplan component is converted into corresponding energy consumption figures.

C. Architectural Occupancy Characterization

For AVF modeling, we collect the time stamps of accesses to micro-architecture structures during basic block characterization. These access times will in turn be used to compute actual occupancy information at source-level simulation time. Access time stamps are computed as the offsets relative to the start of the current basic block.

For register files, the ISS traces are analyzed to obtain the register ID and access offset during pairwise characterizations. As shown in Fig. 3 for register *r29* in block BB_G, together with the duration of each block obtained from timing characterization, the actual access times will be calculated as the sum of the accumulated execution time at the entry of the current block plus the characterized access offset.

The cache occupancy analysis is performed separately in the cache model. Different from register files, the time stamps for data cache accesses are lumped together and estimated at the granularity of basic block boundaries, i.e. all memory accesses are assumed to occur at the beginning/end of a basic block. This avoids the need for target-dependent memory instruction trace analysis, but leads to inherent inaccuracies. Nevertheless, as results will show, such simplifications do not significantly increase AVF estimation errors for caches as compared to register files.

VI. Back-Annotation

Metrics gathered during the characterization step are recorded in a mapping table. As the last step in our flow, this mapping table is then used for directing the annotation of target metrics into the compilable IR at correct insertion points.

A summary of our PERPT annotation flow is shown in Fig. 3. For timing and energy estimation, the annotations into the compilable IR are in the form of global time and energy counters (*CycleCnt* and *EnergyCnt*), global constant arrays (*Delay[][]* and *Energy[][]*) containing delay and energy

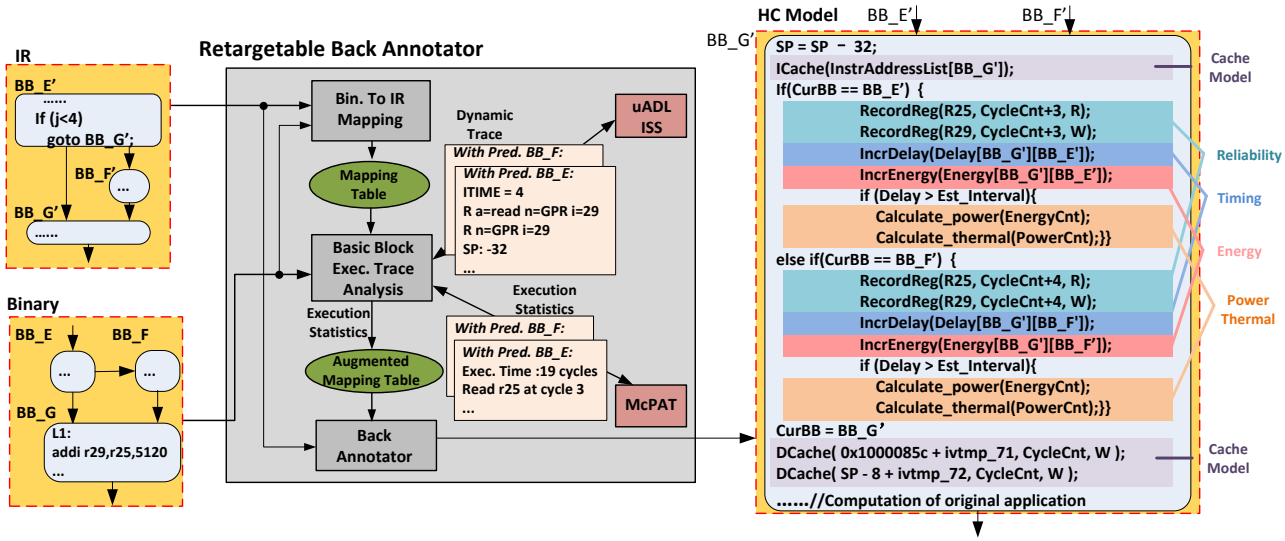


Fig. 3. PERPT back annotation.

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. Counter increments are thereby encapsulated in *incrDelay()* and *incrEnergy()* functions. For any cache access, the latency and energy consumption of a specific access type (read/write) is provided by an annotated cache model. During source-level simulation, data cache latency and energy consumption are added to the global time and energy counters in each call to the *DCache()* function. Timing and energy effects of instruction caches are considered by inserting a similar *ICache()* function call at the beginning of each block. The instruction address trace of the current basic block (*InstrAddressList[]*) is thereby extracted during prior characterization. During simulation, this trace is passed into and interpreted by the *ICache()* model as a trace of consecutive access addresses. Note that since we currently do not estimate instruction cache occupancy, and since all accesses are reads, the access time stamp and type is not needed for *ICache()* calls.

To generate detailed power traces for thermal models, energy consumption of each floorplan component is recorded separately as an array of global energy counters *EnergyCnt[C]* and back-annotated pair-wise energy estimates *Energy[][][C]* over *C* floorplan components. Using these energy estimates, power numbers are calculated and recorded every sampling period (*Calculate_power()*) and fed into a thermal model for temperature estimation (*Calculate_thermal()*). For reliability modeling, register and cache access time stamps and types are annotated with corresponding *RecordReg()* and *DCache()* function calls, which internally invoke an occupancy analyzer for AVF calculation (see Section VI-A).

For the example of annotated code shown in Fig. 3, BB_G is characterized with each of its immediate predecessors, BB_E and BB_F, and two sets of execution metrics are both annotated into the IR. The choice of picking the correct set of annotated metrics is made dynamically during host execution. In this example, if a particular execution of block BB_G follows execution of block BB_F, profiling metrics for predecessor BB_F are picked. As such, a corresponding timing delay and

```
int DCache( int Addr, int AccessCycle, enum AccessType T){
  OutCome = IsCacheHit(Addr);
  UpdateCacheStatus(Addr);

  if (OutCome==MISS){
    CycleCnt += MISS_LATENCY;                          Timing
    if (T==W) EnergyCnt[DCACHE] += W_MISS_ENERGY;      Energy
    if (T==R) EnergyCnt[DCACHE] += R_MISS_ENERGY;
  }
  else if (OutCome==HIT){
    CycleCnt += HIT_LATENCY;                           Timing
    if (T==W) EnergyCnt[DCACHE] += W_HIT_ENERGY;       Energy
    if (T==R) EnergyCnt[DCACHE] += R_HIT_ENERGY;
  }

  RecordCache(Addr, AccessCycle, T, OutCome)           Reliability
}
```

Fig. 4. Lightweight cache model.

energy number is accumulated, optionally for each floor plan component. For register AVF modeling, the access time for $R29$ is estimated as an offset of 4 cycles on top of the current *CycleCnt*. The access type is a *register write*. The cache model is further invoked with reconstructed target addresses at each point of global and stack accesses.

In the following, we will discuss the cache, reliability and thermal sub-models in more detail.

A. Lightweight Cache Model

Since cache behavior is highly dependent on the dynamic execution context, cache-related PERPT estimation and analysis is encapsulated in an online cache model instead of being statically annotated during basic block characterization. Our basic cache mode is similar to previous work [20], [21], [22], [23]. An example is shown in Fig. 4. The cache model is purely behavioral and does not store any actual data. During simulation, it is invoked upon each cache access, and cache hit/miss outcomes are calculated based on the reconstructed address trace and reference cache configuration (*IsCacheHit(Addr)*). With this information, the global timing and energy counters are incremented by corresponding latency and energy values. Cache access latency and energy consumption for different access types are extracted from the ISS description and CACTI/McPAT. Finally, we extend the basic cache model with an occupancy analyzer that is invoked for AVF estimation,

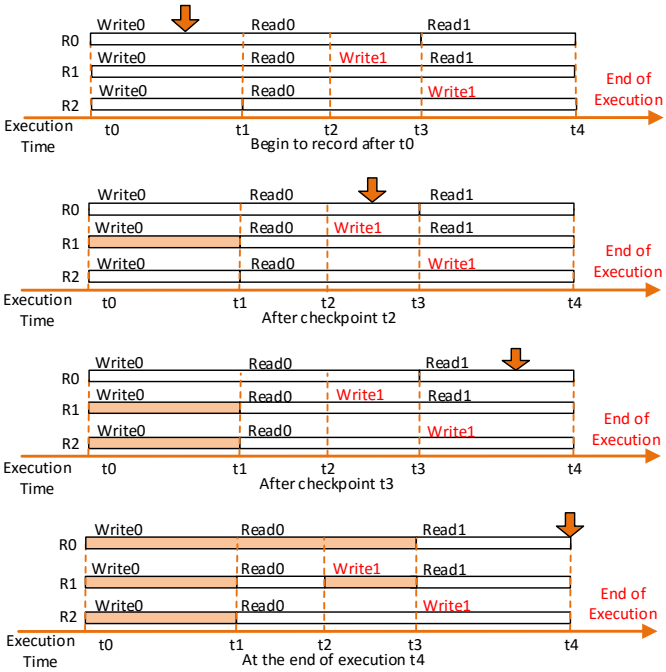


Fig. 5. Producer and Consumer Pair Analysis.

shown as a call to a *RecordCache()* function. Details of the occupancy analyzer will be discussed next.

B. Reliability Modeling

The AVF of a particular data storage structure can be obtained by estimating the occupancy of all the variables waiting to be eventually consumed. A soft error will manifest itself when faulty data is read out of a particular location. The estimation of AVF can be converted into the problem of capturing the variable lifetimes of all resident data. In order to estimate the AVF for data storage structures, we apply a producer-consumer analysis and dynamically re-construct such variable lifetimes during source-level simulation.

1) Register File Analysis: The AVF of each register can be estimated as the ratio of the total sum of time periods when the register is occupied relative to the execution time of the whole program. The overall AVF of the register file can be further calculated as the average occupancy of all its registers over the entire program execution.

Sample register access traces for registers $R0, R1$ and $R2$ are shown in Fig. 5. The key idea in our analysis is to capture the time stamp at the end of each variable's lifetime, which we call *checkpoint* in our following explanation. A checkpoint can either be a write operation to an occupied storage location or the end of the entire program execution. In both cases, the existing data in the access location is guaranteed to not be consumed again, and the variable lifetime can be calculated. In this way, the architectural occupancy of a certain storage structure is divided into atomic variable lifetime periods defined by these check points.

An example of our producer-consumer analysis is shown in Fig. 5. The execution begins with initial write operations to registers $R0, R1$ and $R2$. During execution, $R1$ and $R2$ are written again at times $t2$ and $t3$, respectively, which are highlighted in red as the checkpoints for both registers. Finally, the end of the execution at time $t4$ will be the checkpoint for all registers. During simulation, latest access times of all write and read operations are continuously recorded and updated. For each storage unit, whenever a checkpoint is reached, the time difference between the latest read and write will be calculated and accumulated as the occupancy duration. At the final checkpoint at the end of simulation, occupancy times of all storage cells can be obtained and used for AVF estimation.

In Fig. 5, the start time $t0$ is first recorded as the latest write time stamp for all registers. At $t1$, the latest read time for all registers is recorded as $t1$. At time $t2$, $R1$ then reaches its checkpoint, and the difference between $t1$ and $t0$ is accumulated as $R1$'s occupancy duration, represented as shaded bar in Fig. 5. At time $t3$, $R2$ reaches its checkpoint and the difference between $t1$ and $t0$ is accumulated as its occupancy time. Furthermore, the latest read time for $R0$ and $R1$ is updated to $t3$. At the end of the execution, all registers hit their checkpoint. The period between $t0$ and $t3$ is recorded as $R0's$ occupancy duration, and the period between $t2$ and $t3$ is added to $R1$'s occupancy. For $R2$, since there is no subsequent read operation after $t3$, the write operation at $t3$ is discarded. In this way, as shown by the shaded bars in the figure, the occupancy periods of all registers are captured at the end of execution.

2) Data Cache Analysis: For registers, checkpoints include all writes and the end of the program execution. By contrast, for caches, there are four different access types from the processor's perspective: *write miss, write hit, read miss* and *read hit*. Besides write operations, a cache miss on read can also mark the beginning of a new cache data lifetime, since a new cache line will be filled into the cache. Hence, checkpoints for the producer-consumer analysis of data caches include write hits and misses, read misses, and the end of program execution. Upon each such event, the occupancy of each cache entry can be calculated and accumulated. The data resident in the access location is either overwritten or evicted, and the old data is guaranteed to not be consumed again. After identifying all such checkpoints, a similar analysis as for register files is applied to cache AVF estimation.

3) AVF Estimation: The implementation of producer-consumer analysis within our overall source-level simulation framework is shown in Algorithm 1. For register files, a global *RegFile* array is introduced to record the register file access history. For each register, the array is used to store the latest write time, the latest read time and the total accumulated occupancy time. As discussed earlier in Section VI, the function *RecordReg()* is inserted into each basic block during back annotation. During simulation, *RecordReg()* will be called to monitor and record each register access. Internally, the function applies the aforementioned producer-consumer analysis to update corresponding access and occupancy time information in the global *RegFile* array. Finally, at the end of execution and hence the final checkpoint, all the remaining occupancy times for every register are collected.

For the data cache, a global *DCArr* array is used to record the latest cache read and write miss time, the latest read hit time and the total accumulated occupancy time for each

Algorithm 1 Occupancy analyzer.

```
1: function RECORDREG(RegID, Cycle, Type)
2:     if Type == W then
3:         RegFile[RegID][ACC]+=
4:         (RegFile[RegID][LAST_R]-RegFile[RegID][LAST_W]);
5:         RegFile[RegID][LAST_R] = Cycle;
6:         RegFile[RegID][LAST_W] = Cycle;
7:         return
8:     end if
9:     if Type == R then
10:        RegFile[RegID][LAST_R] = Cycle;
11:    end if
12: end function
13:
14: function RECORDCACHE(Addr, Cycle, Type, Outcome)
15:    if (Type == W) or (Type == R and Outcome == Miss) then
16:        DCArr[Addr][ACC]+=
17:        (DCArr[Addr][LAST_R]-DCArr[Addr][LAST_W]);
18:        DCArr[Addr][LAST_R] = Cycle;
19:        DCArr[Addr][LAST_W] = Cycle;
20:        return
21:    end if
22:    if Type == R and Outcome == Hit then
23:        DCArr[Addr][LAST_R] = Cycle;
24:    end if
25: end function
```

word in the cache. During execution, memory access traces generated by the source-level model are fed into a cache access recorder, shown as *RecordCache()* in Algorithm 1. As discussed before, checkpoints in the cache analysis are also dependent on cache hit and miss information. Before performing the producer-consumer analysis, using the cache model, the cache access outcome therefore needs to be computed and passed into the *RecordCache()* function. Upon each cache write and read miss, i.e. on every checkpoint, the occupancy time will be accumulated. Otherwise, if a cache access is a read hit, the latest read time will be updated.

C. Thermal Modeling

We support integration of two thermal models, HotSpot and DTTEM, into our source-level simulation to generate transient and steady-state thermal profiles.

1) HotSpot: HotSpot [36] dynamically builds a representative RC model of the chip based on a given floorplan, and it computes the temperature profile over a sequence of time stamps (called sampling periods) based on a given trace of power dissipation values. HotSpot requires a structurally accurate power trace over all components in the floorplan at the chosen sampling granularity. We generate these traces using the extended characterization approach described in the previous section (shown as calls to the *Calculate_power()* function). Within the *Calculate_thermal()* function, the power values are written to a Unix pipe created at the start of simulation. We have modified HotSpot to accept data streamed through a pipe and interpret the written power values to calculate the temperature.

2) DTTEM: DTTEM [41] [42] uses similar concepts as HotSpot for temperature estimation. This model requires conductance ($\mathbf{G}$) and capacitance ($\mathbf{C}$) matrices, which are characteristic of a chip floorplan. To maintain consistency between DTTEM and HotSpot thermal models, and without loss of generality, we extract RC representations from HotSpot and use MATLAB to generate the parameters required by DTTEM. Any assumptions or approximations made during generation of RC models in HotSpot are thus retained by DTTEM.

The primary difference between DTTEM and HotSpot is the temperature evaluation mechanism. DTTEM assumes that, with sampling of power values at small and constant time intervals, transient temperature evaluation can be discretized. This assumption leads to a simplified solution of the basic first-order heat transfer differential equation. We integrate this solution into our flow to achieve faster temperature estimation [3].

VII. Experimental Results

We implemented our automated, retargetable back-annotation (RBA) flow in Python using the uADL ISS [5], McPAT [26] and HotSpot [36] as PERPT references. The RBA tool is available for download in open-source form at [44]. To evaluate our flow, 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.

Six benchmarks from the MiBench suite [45] with both small and large data sets were selected for validation. Among those, FFT followed by SHA have the largest static code size in terms of both instructions and basic blocks. This leads to larger back-annotation runtimes. ADPCM has the smallest average basic block size, which affects simulation overhead and accuracy. CRC32 has the smallest code size, but a relatively large memory footprint with 30% of dynamically executed instructions being memory accesses. The memory footprint determines overhead of cache simulations. Stringsearch (StrSrch) and ADPCM have the largest and smallest footprints of 35% and 22%, respectively. The FFT benchmark uses floating-point emulation on our PowerPC targets. We inlined all such emulation calls into respective basic blocks during characterization. This can lead to additional inaccuracies in our flow. In case of other library calls, since back-annotation requires source code, we excluded benchmarks for which library sources are not available. Benchmarks were further modified to validate proper function call characterization [2].

We evaluated source-level simulation for an in-order, e200_z4-like dual-issue core with 16KB 4-way associative L1 data cache and instruction cache, and an e200_z6-like single-issue core with a longer pipeline. The two targets are representative of typical embedded processor micro-architectures. Note that the target ISA thereby has little influence on simulation accuracy and speed. Both targets do not include floating point units or dynamic branch predictors. For power and thermal estimations, we assume a 500MHz operating frequency. A gcc-4.4.5 cross-compiler with -O2 optimization level is used to generate executable binary files.

A summary of experimental results for both targets are shown in Table I and II. For performance, energy and reliability, results of source-level (SL) estimation with our proposed cache model are compared against cycle-accurate ISS/McPAT

TABLE I
HOST-COMPILED AND REFERENCE ESTIMATION RESULTS FOR Z4

Benchmark	Timing [cycles]		Energy [mJ]		Reliability [AVF]				Power [W]		Thermal [K]		
					Register		D-Cache				SL		Ref.
	SL	Ref.	SL	Ref.	SL	Ref.	SL	Ref.	SL	Ref.	HS	DTTEM	
SHA (Sm.)	24,749,050	24,705,419	13.8	13.6	0.36	0.36	0.59	0.59	0.279	0.277	319.48	319.47	319.48
SHA (Lg.)	257,729,762	258,144,360	143.4	141.4	0.43	0.43	0.61	0.61	0.278	0.277	319.48	319.47	319.48
ADPCM (Sm.)	75,714,673	77,711,244	40.3	40.1	0.51	0.55	0.6	0.59	0.267	0.258	319.42	319.41	319.38
ADPCM (Lg.)	1,464,587,108	1,501,704,686	784.9	780.2	0.52	0.55	0.57	0.58	0.268	0.259	319.42	319.42	319.39
CRC32 (Sm.)	27,420,147	26,054,375	14.3	13.6	0.24	0.24	0.87	0.88	0.261	0.270	319.41	319.41	319.47
CRC32 (Lg.)	533,110,925	506,501,540	277.3	264.3	0.24	0.24	0.88	0.88	0.261	0.270	319.41	319.41	319.46
Dijkstra (Sm.)	74,174,937	75,634,298	33.5	33.7	0.30	0.31	0.89	0.87	0.226	0.227	319.26	319.26	319.27
Dijkstra (Lg.)	375,064,414	382,441,510	169.6	173.6	0.30	0.31	0.89	0.87	0.226	0.227	319.26	319.26	319.27
FFT (Sm.)	145,677,833	157,738,762	76.4	83.8	0.52	0.53	0.86	0.84	0.257	0.265	319.29	319.29	319.50
FFT (Lg.)	1,342,443,584	1,419,826,338	689.7	754.4	0.52	0.53	0.86	0.84	0.256	0.265	319.29	319.29	319.50
StrSrch (Sm.)	1,031,753	977,657	0.4	0.4	0.06	0.06	0.05	0.05	0.210	0.227	319.22	319.22	319.31
StrSrch (Lg.)	23,554,554	22,317,149	9.9	10.1	0.06	0.06	0.50	0.47	0.210	0.227	319.22	319.22	319.31

TABLE II
HOST-COMPILED AND REFERENCE ESTIMATION RESULTS FOR Z6

Benchmark	Timing [cycles]		Energy [mJ]		Reliability [AVF]				Power [W]		Thermal [K]		
					Register		D-Cache				SL		Ref.
	SL	Ref.	SL	Ref.	SL	Ref.	SL	Ref.	SL	Ref.	HS	DTTEM	
SHA (Sm.)	35,983,039	36,429,267	16.5	16.7	0.37	0.37	0.59	0.59	0.232	0.229	319.56	319.57	319.56
SHA (Lg.)	375,326,029	379,980,280	172.1	173.8	0.44	0.44	0.61	0.61	0.230	0.229	319.56	319.58	319.56
ADPCM (Sm.)	119,584,284	115,866,610	51.0	49.1	0.51	0.55	0.57	0.59	0.215	0.212	319.47	319.48	319.46
ADPCM (Lg.)	2,324,832,871	2,253,595,320	994.7	956.6	0.52	0.55	0.58	0.58	0.215	0.213	319.47	319.49	319.47
CRC32 (Sm.)	41,111,640	41,066,243	17.6	17.7	0.24	0.24	0.88	0.88	0.215	0.216	319.47	319.48	319.49
CRC32 (Lg.)	799,222,969	798,336,362	342.2	344.1	0.24	0.24	0.88	0.88	0.215	0.216	319.47	319.48	319.48
Dijkstra (Sm.)	108,090,303	109,560,552	41.8	42.3	0.30	0.31	0.89	0.87	0.193	0.193	319.35	319.36	319.35
Dijkstra (Lg.)	546,557,390	553,991,012	211.3	214.1	0.30	0.31	0.89	0.87	0.193	0.193	319.35	319.36	319.35
FFT (Sm.)	207,466,663	233,253,571	86.1	96.1	0.52	0.53	0.86	0.84	0.208	0.221	319.36	319.37	319.55
FFT (Lg.)	1,909,751,431	2,103,216,994	803.0	931.3	0.52	0.53	0.86	0.84	0.209	0.221	319.36	319.37	319.55
StrSrch (Sm.)	1,405,434	1,464,203	0.5	0.6	0.06	0.06	0.05	0.05	0.189	0.192	319.32	319.33	319.36
StrSrch (Lg.)	32,071,425	33,410,850	11.9	12.8	0.06	0.06	0.50	0.47	0.187	0.192	319.32	319.33	319.36

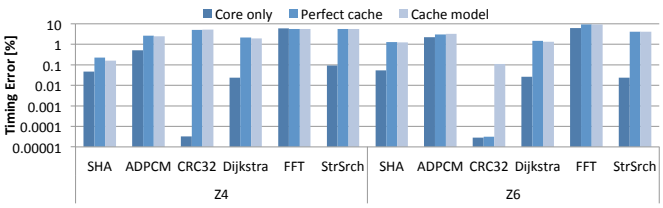


Fig. 6. Source-level timing estimation accuracy.

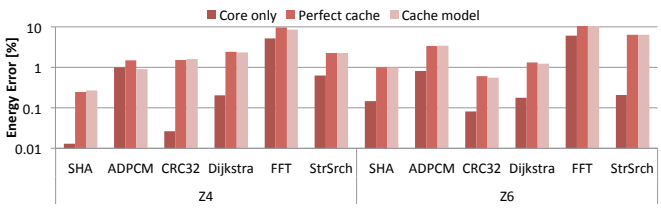


Fig. 7. Source-level energy estimation accuracy.

reference (Ref.). For power estimations, average power over whole program executions is compared. For source-level thermal estimations, average steady-state temperatures over all floorplan components using either HotSpot (HS) or DTTEM models are summarized. Likewise, for reliability, average AVF estimates across all registers and cache lines are shown. Detailed breakdowns of power, thermal and reliability accuracy, including transient power and thermal results will be discussed in the following sub-sections. Note that source-level accuracies are very similar between small and large data sets. Unless noted otherwise, only results for large data sets will be shown.

A. Timing and Energy Results

Fig. 6 and 7 compare the accuracy of timing and energy results for complete application runs of various benchmarks. In order to explore tradeoffs in cache modeling, we investigate three cases: (1) estimating timing and energy for the processor core only excluding caches (*Core only*), (2) modeling all memory accesses as perfect caches with 100% hit rate (*Perfect cache*), and (3) incorporating our online cache model as described in Section VI-A (*Cache model*).

For timing simulation (Fig.6), the maximum timing estimation errors on the Z4 target are 5.8% for the core only, 5.4% for a perfect cache and 5.4% for an accurate cache model, while average errors are 1.1%, 3.5% and 3.5%, respectively. On the Z6 target, the maximum estimation errors are 6.2% for the core only, 9.2% for a perfect cache and 9.2% for an accurate cache model, while average errors are 1.4%, 3.2% and 3.2%, respectively.

The energy estimation results (Fig.7) largely depend on the timing accuracy, where the maximum estimation errors on the Z4 target are 5.1% for core only, 9.7% for the perfect cache and 8.6% for the accurate cache model, while average errors

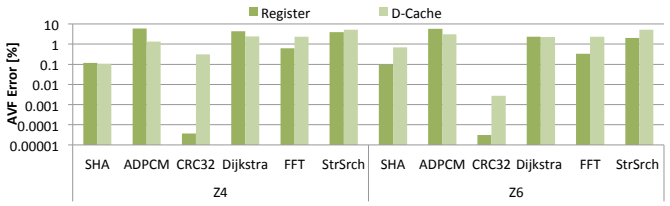


Fig. 8. Source-level AVF estimation accuracy.

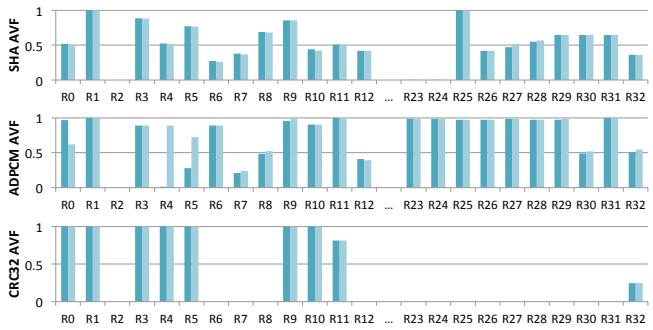


Fig. 9. Occupancy of each individual register for Z4 target and small input data set.

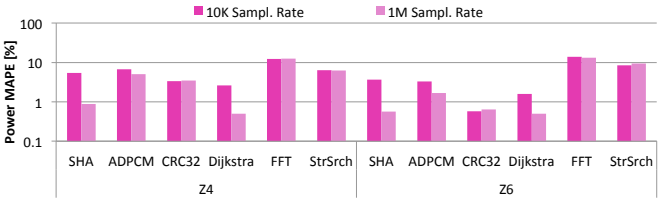


Fig. 10. Source-level transient power estimation accuracy.

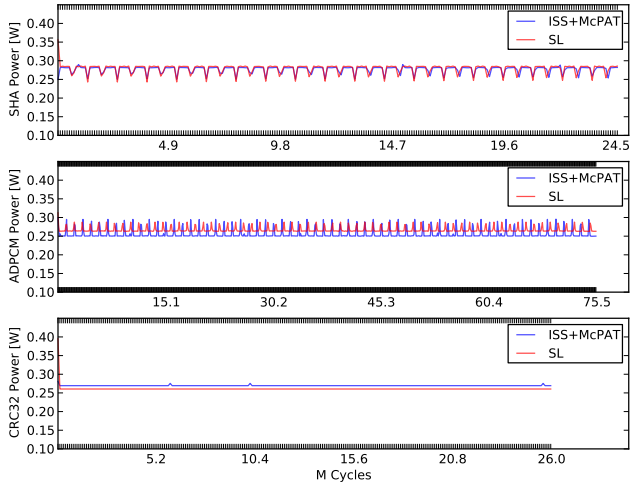


Fig. 11. Transient power traces on Z4 target.

are 1.2%, 2.9% and 2.7%, respectively. On the Z6 target, the maximum estimation errors are 6.0% for core only, 10.0% for a perfect and 10.0% for an accurate cache, while average errors are 1.2%, 3.9% and 3.9%.

Overall, timing and energy estimation accuracy is more than 90% in all cases. The FFT as the only floating-point benchmark shows the largest errors. Since inlined floating-point emulation routines can be data dependent, additional errors are introduced when statically estimating their timing and energy. This reflects a limitation of our approach. The ADPCM benchmark exhibits relatively larger errors since its IR has comparatively smaller blocks with a larger number of consecutive branches. This leads to pipeline dependencies that span across more than two blocks, which are not accurately captured by our pair-wise characterization. There are multiple sources of inaccuracies for timing and energy estimations when caches are included. As mentioned before, to keep our cache models lightweight, a fixed number of cycles are attached to each cache miss. In reality, however, miss penalties can vary dynamically. For example, since the memory port in our target is 32-bit wide, it may take multiple cycles to fetch data into the 32 byte line-fill buffer. Hence, consecutive cache misses may interfere, and a later cache access may need to wait for additional cycles until an earlier cache miss has read the fetched data from the buffer. For these reasons, the StrSrch and CRC32 benchmarks, which have the largest memory footprints, result in larger estimation errors for simulations with both perfect and accurate cache models.

Overall results show that inclusion of caches will incur an average 1.7% inaccuracy compared to modeling the core only. Most embedded benchmarks have relatively low cache miss rates under typical cache configurations [45], [21], [22], [23]. As such, simulations assuming a perfect cache only decrease accuracy by 0.1% on average compared to an accurate cache model. As will be shown in Section VII-E, despite being lightweight, the slightly better accuracy of an accurate cache model thereby comes with a significant speed penalty. Furthermore, in cases where over-estimated core cycle counts are cancelled out by cache under-estimations (see CRC32 example), the perfect cache model can actually have a lower error rate.

B. AVF Estimation Results

We simulate the occupancy of each register and cache line and calculate the register file and data cache AVF as the average occupancy of each storage location throughout each whole program execution. We collect reference estimations by parsing and analyzing detailed cycle-accurate uADL simulator output traces to compare against our source-level simulation results.

Fig. 8 shows the register file and data cache AVF estimation accuracy. Generally, estimation errors are mainly due to timing inaccuracies manifesting themselves as jitter in recorded access times. For the register file, the largest occupancy errors are 6.01% for the Z4 and 5.83% for the Z6 target, while the average errors are 2.50% and 1.77%, respectively. The largest error is seen in the ADPCM benchmark. Again, this is due to inaccuracy in capturing timing dependencies spanning across more than two blocks. For data caches, the largest occupancy errors are 5.26% for the Z4 and 5.26% for the Z6 target, while the average errors are 1.95% and 2.27%, respectively.

Fig. 9 shows the breakdown of AVFs for individual registers for the SHA, ADPCM and CRC32 benchmarks. Registers $R13$ to $R22$ are reserved, and their occupancy is always 0%. As such, they are omitted in Fig. 9. In the PowerPC calling convention, registers $R23$ to $R31$ are usually used as link registers to store return addresses. Hence, they will

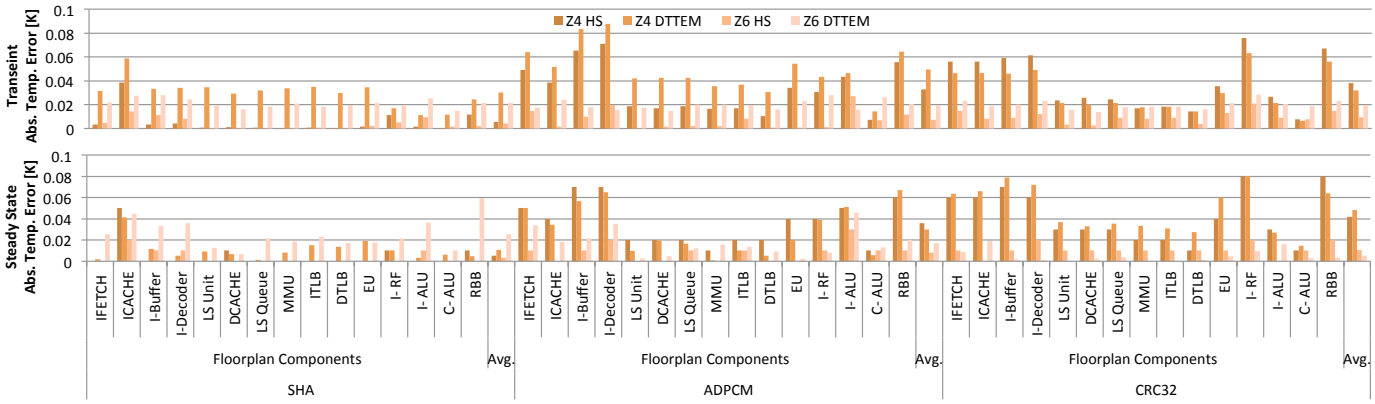
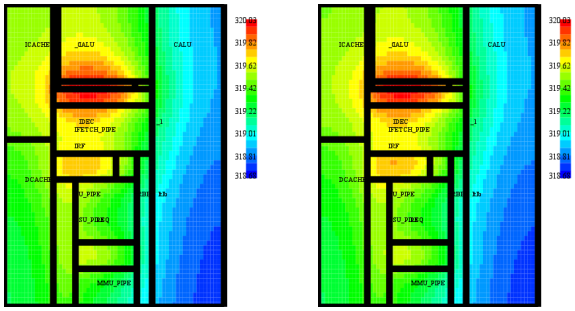
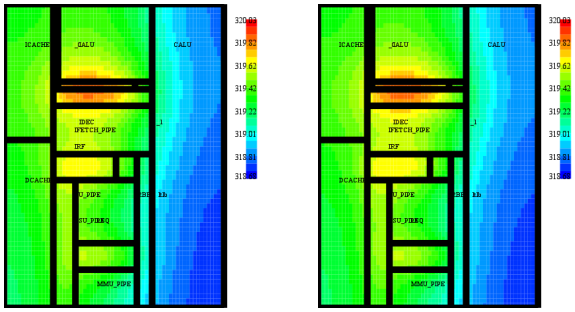


Fig. 12. Average absolute transient and steady-state temperature errors.



(a) Source-level (b) Reference

Fig. 13. Steady-state hotspot formation in SHA (Z6).



(a) Source-level (b) Reference

Fig. 14. Steady-state hotspot formation in CRC32 (Z6).

only be occupied when there are large numbers of function calls during the execution, which is the case in the ADPCM and SHA benchmarks. CRC32 consists of a single function only, and link registers remain unused. Results show that the source-level simulation can accurately replicate the dynamic occupancy of each register, with less than 1% error on average. A similar breakdown is possible for vulnerability of individual cache lines. This shows the ability of our approach to enable detailed soft-error reliability analysis at fine structural granularity.

C. Transient Power Results

Fig. 10 shows the mean absolute percentage error (MAPE) of the transient power traces obtained from source-level models as compared against the cycle-accurate reference. In the reference flow, similar scripts as in the back-annotation are used to identify latency information and access statistics for different sampling intervals. For each sampling interval, McPAT is invoked to obtain power values. We choose two different sampling periods, 10K and 1M cycles, to evaluate our flow. Caches are included, and an accurate source-level cache model is used.

Under a 10K sampling rate, the maximum MAPE on the Z4 and Z6 targets is 12% and 13%, while average errors are 6% and 5%, respectively. Under 1M cycles sampling, maximum MPAEs are 12% and 13%, while average errors are 4% and 4% for the Z4 and Z6 targets, respectively. Due to averaging effects, our infrastructure shows a better estimation accuracy under a larger sampling period. Inaccuracies of source-level power models mainly stem from errors in the underlying basic timing and energy estimates.

Fig. 11 compares the source-level (SL) and reference power traces for SHA, ADPCM and CRC32 examples under a 10K cycles sampling period for small input data sets on a Z4 target. Data intensive applications with simple control flow, such as CRC32, will demonstrate a constant power value for most of the execution time in both models. The power trace from source-level estimation of the SHA example tightly follows the reference trace, while ADPCM and CRC32 show larger transient or constant mismatches due to the relatively larger timing and energy errors discussed in previous sections.

D. Thermal Estimation Results

We perform thermal simulations on Z4 and Z6 architectures with a floorplan area (for 90 nm process technology) of 4.77 mm^2 and 6.85 mm^2, respectively. The ambient temperature is assumed to be 318.15 K. A 1M cycles sampling period is chosen as a tradeoff between precision and overhead. As reference flow for all comparisons, the dynamic power trace from the cycle-accurate reference models is fed into HotSpot.

To calculate transient and steady-state temperatures, HotSpot is configured to generate a temperature profile using a block-based model. HotSpot reports both types of temperature estimations for all the components of a floorplan. Similar values are obtained from DTTEM. Transient errors for both models are measured as average absolute errors over all sampling periods.

Fig. 12 shows the absolute transient and steady-state error of source-level thermal simulations for SHA, ADPCM and CRC32 benchmarks across all models and processors. For transient estimation, the maximum error is 0.08K for the Z4

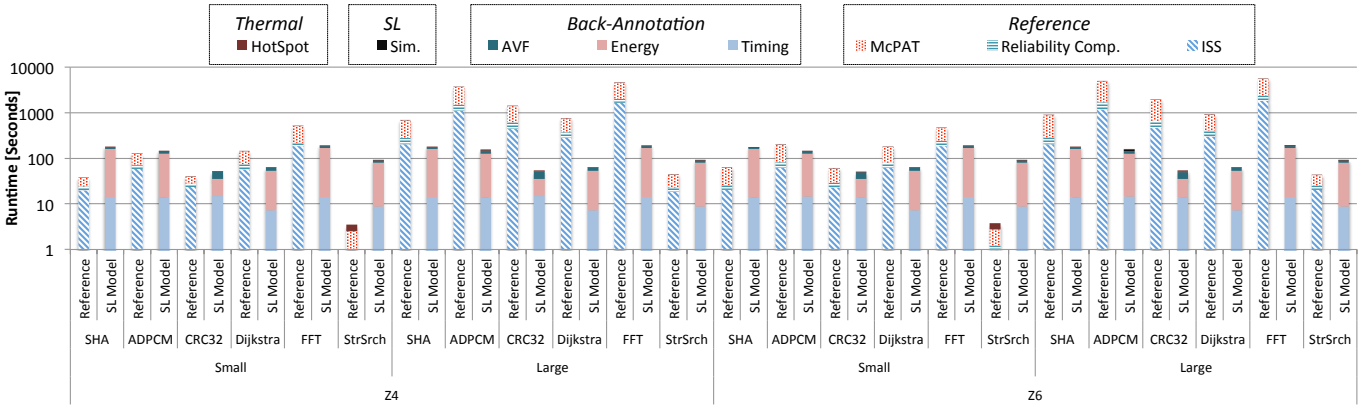


Fig. 15. Total run time comparison.

processor and 0.03K for the Z6 processor using a HotSpot-based model. In case of DTTEM, maximum errors are 0.09K and 0.03K for the Z4 and Z6 targets, respectively. The average absolute transient errors of a HotSpot-based model are 0.03K and 0.007K for the Z4 and Z6, and 0.04K and 0.02K for DTTEM. For steady-state estimations, when HotSpot is used as the thermal model, the maximum error is 0.08K for the Z4 processor and 0.03K for the Z6 processor. The maximum errors in case of DTTEM are 0.08K and 0.06K, respectively. The average absolute steady-state errors in both models are low. Usage of DTTEM instead of HotSpot increases the average steady-state error only marginally (0.015K vs. 0.007K for a 0.008K increase in the Z6, and an increase by 0.002K from 0.028K to 0.030K in the Z4). Errors are higher in case of DTTEM mainly because of the approximations related to discretization. For example, it cannot capture the secondary effects of current temperature values on the next set of temperatures.

We further analyze steady-state thermal profile generation using a grid-based HotSpot with a grid size of 64x64. The temperature maps of SHA and CRC32 on the Z6 target are shown in Fig. 13 and Fig. 14. The hotspots identified by both flows are in the instruction fetch block and the execution pipelines. Including caches, we can observe that our flow can accurately track temperature variations across different benchmarks with minimal differences in steady-state temperature profiles.

E. Simulation Speed

We evaluated our source-level PERPT infrastructure by comparing the total runtime and simulation throughput against the reference flow.

Fig. 15 shows the runtime breakdown of the reference and source-level modeling flows. In the thermal case, we used HotSpot-based source-level models to establish a fair comparison. As will be shown later, the DTTEM-based model has a similar runtime and speedup. A 1M-cycle period is picked as a common sampling rate for power and thermal estimations.

The total runtime of the reference flow includes the ISS timing simulation, power and energy estimations by McPAT, AVF estimation using ISS execution traces and thermal estimation by invoking HotSpot. In Fig. 15, additive contributions

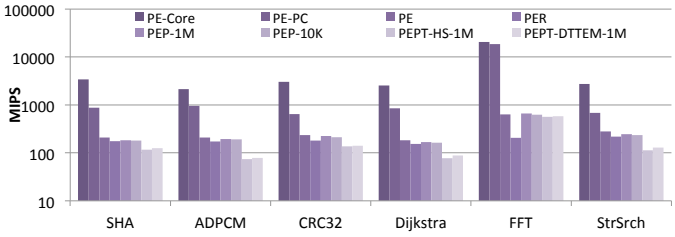


Fig. 16. Source-level PERPT simulation throughput.

to overall runtime are highlighted as *ISS, Reliability Comp., McPAT* and *HotSpot*. For the Z4 target, an average of 2.3 minutes of execution time are required for running the whole flow over small data sets, while large data sets require 32.0 minutes on average. For the Z6 target, the average runtimes are 2.6 minutes and 41.1 minutes, respectively. Cycle-accurate ISS execution and power trace generation contribute more than 85% to the overall reference runtime. This is because detailed execution traces need to be generated in the ISS simulation and McPAT needs to be invoked repeatedly to compute transient power traces. We excluded trace parsing and file I/O overhead in the reference flow to establish a fair comparison.

In case of source-level models, the runtime consists of back-annotation time (*Timing, AVF, Energy*), source-level models simulation time (*SL/Sim.*) and HotSpot execution time (*HotSpot*). For the Z4 target, average runtimes for full PERPT estimation are 1.9 minutes for small and 2.1 minutes for large data sets. On the Z6 target, 2.1 minutes and 3.0 minutes are required, respectively. During back annotation, timing, energy and AVF metrics of each basic block pair have to be obtained from reference models. As such, timing and energy characterizations, which require detailed cycle-accurate execution traces and McPAT invocations contribute the most to overall overhead, taking more than 71% of the total runtime on average. By contrast, extending the characterization to also extract AVF time stamps contributes little additional complexity.

Time spent in back-annotation depends on code size and control flow graph complexity. In applications with large code size but low dynamic instruction counts, there is more relative overhead in total runtime versus a reference flow. This is the case for small data sets and the StrSrch example, for

which the source-level flow does not provide a better runtime. However, back-annotation is a one-time effort. Its overhead will be outweighed by the additional execution time for large input data sets. The source-level model's execution time does not significantly scale with input size, and it demonstrates up to 15-20 times faster execution speeds in such cases. Furthermore, once generated, a back-annotated source-level model can be repeatedly resimulated under different input scenarios.

Finally, Fig. 16 shows the simulation throughput of back-annotated PERPT source-level models for large data sets. Simulation throughput is measured in target-equivalent MIPS, calculated as the ratio of dynamic instructions executed on the target over the execution time of the source-level model. We do not count back-annotation time in the calculation of throughput, as this is a one-time effort for a given benchmark. For timing and energy estimation alone, average speeds of 5740 MIPS, 3780 MIPS and 290 MIPS can be achieved when estimating the core only (PE-Core), assuming a perfect cache (PE-PC) or incorporating an accurate cache model (PE), respectively. When also including reliability, an average speed of 185 MIPS is achieved for a model that performs register file and D-Cache AVF estimation (PER). For power estimation (PEP), power traces can be generated with 1M and 10K sampling rates at 280 and 268 MIPS on average. Finally, Hotspot- and DTTEM-based thermal models with 1M sampling rate (PEPT-HS-1M and PEPT-DTTEM-1M) provide 180 and 191 MIPS throughput on average, respectively. Overall, simulation throughputs are several orders of magnitude faster than an equivalent ISS execution with an average of only 0.658 MIPS. If time-consuming execution trace generation and parsing are included, reference throughputs can be even lower.

Multiple factors influence simulation speeds. Larger basic block sizes will result in higher core-only base speeds, as more target instructions can be simulated with lower branching overhead. The FFT benchmark with inlined floating-point emulations has a much larger simulation throughput for that reason. It further benefits from the fact that floating-point operations requiring a large number of emulated target instructions can be natively executed using hardware support on the host. Among other benchmarks, ADPCM has the smallest average block and lowest base speed. The memory footprint determines simulation overhead when introducing cache models. A larger footprint will result in more invocations of the cache. This is the case for StrSrch and CRC32 benchmarks, which are the slowest among all models with perfect cache. Note that an accurate cache model will reduce simulation speed significantly. This establishes a tradeoff between accuracy and speed of different modeling levels. Variations in speed of accurate caches thereby also depend on specific access patterns and cache outcomes. Finally, incorporating reliability, power and thermal models incurs additional overhead. Specifically, reliability modeling requires back-annotation of individual target register accesses. This is especially pronounced in the FFT benchmark with a large ratio of (emulated) target instructions per source-level operation. By contrast, power and thermal annotations add a fixed overhead on top of basic timing and energy models.

VIII. Summary and Conclusion

In this paper, we propose a novel multi-metric source level simulation infrastructure for performance, energy, reliability, power and thermal (PERPT) estimations. We leverage existing retargetable architecture description language (ADL) frameworks for basic block timing and energy characterization. For reliability modeling, we evaluate the register file and data cache vulnerabilities against soft errors by estimating the AVF using an online occupancy analyzer. Finally, the host-compiled source-level model is integrated with thermal models for fast temperature estimation. The one-time back-annotation of code is fast (on the order of 1-2 minutes), while resulting models are more than 90% accurate for timing, energy, reliability and power estimation, with an average error of 0.05K for steady-state thermal estimation. Models are orders of magnitude faster than existing reference flows, with simulation speeds ranging from 180 MIPS to more than 5740 MIPS. Our back-annotation flow is fully automated. The retargetable back-annotation (RBA) tool is available for download at [44].

IX. Acknowledgements

This research is supported by SRC Tasks 2085.001 and 2317.001 and NSF grant CSR-1421642.

References

[1] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems," in *Design, Automation and Test in Europe (DATE)*, 2015.

[2] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, retargetable back-annotation for host compiled performance and power modeling," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[3] D. Gandhi, A. Gerstlauer, and L. John, "Fastspot: Host-compiled thermal estimation for early design space exploration," in *International Symposium on Quality Electronic Design (ISQED)*, 2014.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[5] The architectural description language project, ver 2.0.0. [Online]. Available: http://opensource.freescale.com/fsl-oss-projects

[6] M. Gligor, N. Fournel, and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible mpsoc simulation," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2009.

[7] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1994.

[8] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$sim: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2008.

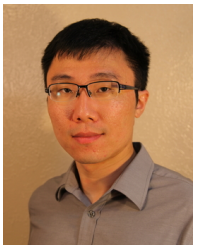
[9] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[10] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[11] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty, and A. Herkersdorf, "High-level timing analysis of concurrent applications on mpsoc platforms using memory-aware trace-driven simulations," in *VLSI System on Chip Conference (VLSI-SoC)*, 2010.

[12] Y. Yi, D. Kim, and S. Ha, "Fast and accurate cosimulation of mpsoc using trace-driven virtual synchronization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 12, pp. 2186–2200, 2007.

[13] Z. Wang and J. Henkel, "Accurate source-level simulation of embedded software with respect to compiler optimizations," in *Design, Automation and Test in Europe (DATE)*, 2012.
[14] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *Design Automation Conference (DAC)*, 2011.
[15] E. Y. Hwang, S. Abdi, and D. Gajski, "Cycle approximate retargettable performance estimation at the transaction level," in *Design, Automation and Test in Europe (DATE)*, 2008.
[16] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic instrumentation of embedded software for high levelhardware/software co-simulation," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009.
[17] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann, "Control-flow-driven source level timing annotation for embedded software models on transaction level," in *Digital System Design (DSD)*, 2011.
[18] R. Plyaskin and A. Herkersdorf, "Context-aware compiled simulation of out-of-order processor behavior based on atomic traces," in *International Conference on VLSI and System-on-Chip (VLSI-SoC)*, 2011.
[19] K.-L. Lin, C.-K. Lo, and R.-S. Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.
[20] A. Pedram, D. Craven, A. Tileli, and A. Gerstlauer, "Modeling cache effects at the transaction level," in *Analysis, Architectures and Modeling of Embedded Systems, International Embedded Systems Symposium (IESS)*, 2009.
[21] Z. Wang and J. Henkel, "Fast and accurate cache modeling in source-level simulation of embedded software," in *Design, Automation Test in Europe (DATE)*, 2013.
[22] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann, "Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2013.
[23] H. Posadas, L. Diaz, and E. Villar, "Fast data-cache modeling for native co-simulation," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011.
[24] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-level timing annotation and simulation for a heterogeneous multiprocessor," in *Design, Automation and Test in Europe (DATE)*, 2008.
[25] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *nternational symposium on Computer architecture (ISCA)*, 2000.
[26] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture (MICRO)*, 2009.
[27] Y.-H. Park, S. Pasricha, F. J. Kurdahi, and N. Dutt, "A multi-granularity power modeling methodology for embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 4, pp. 668–681, 2011.
[28] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto, "A multi-level strategy for software power estimation," in *International Symposium on System Synthesis (ISSS)*, 2000.
[29] C. Brandolese, S. Corbetta, and W. Fornaciari, "Software energy estimates based on statistical characterisation of intermediate compilation code," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2011.
[30] D. Calvo, P. González, L. Díaz, H. Posadas, P. Sánchez, E. Villar, A. Acquaviva, and E. Macii, "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.
[31] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *International Symposium on Microarchitecture (MICRO)*, 2003.
[32] X. Fu, J. Poe, T. Li, and J. A. B. Fortes, "Characterizing microarchitecture soft error vulnerability phase behavior," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.
[33] L. Duan, B. Li, and L. Peng, "Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2009.
[34] J. Carretero, E. Herrero, M. Monchiero, T. Ramírez, and X. Vera, "Capturing vulnerability variations for register files," in *Design, Automation and Test in Europe (DATE)*, 2013.
[35] A. Nair, S. Eyerman, L. Eeckhout, and L. John, "A first-order mechanistic model for architectural vulnerability factor," in *International Symposium on Computer Architecture (ISCA)*, 2012.
[36] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *International Symposium on Computer Architecture (ISCA)*, 2003.
[37] S. Eratne, C. Romo, E. John, and W.-M. Lin, "Reducing thermal hotspots in multi-core processors using dynamic core scheduling," in *International Conference on Computer Design (CDES)*, 2011.
[38] P. Liu, Z. Qi, H. Li, L. Jin, W. Wu, S.-D. Tan, and J. Yang, "Fast thermal simulation for architecture level dynamic thermal management," in *International Conference on Computer-Aided Design (ICCAD)*, 2005.
[39] Y. Han, I. Koren, and C. M. Krishna, "TILTS: A fast architectural-level transient thermal simulation method," *Journal of Low Power Electronics*, vol. 3, no. 1, pp. 13–21, 2007.
[40] Y. Yang, Z. P. Gu, C. Zhu, R. P. Dick, and L. Shang, "ISAC: Integrated space-and-time-adaptive chip-package thermal analysis." *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 86–99, 2007.
[41] L. Thiele, L. Schor, H. Yang, and I. Bacivarov, "Thermal-aware system analysis and software synthesis for embedded multi-processors," in *Design Automation Conference (DAC)*, 2011.
[42] A. Krum, *Thermal Management. In F. Kreith, editor, The CRC Handbook of Thermal Engineering*. CRC Press, 2000.
[43] A. Faravelon, N. Fournel, and F. Pétrot, "Fast and accurate branch predictor simulation," in *Design, Automation Test in Europe (DATE)*, 2015.
[44] "RBA Version 2.0," http://www.ece.utexas.edu/~gerstl/releases/.
[45] Mibench version 1.0. [Online]. Available: http://www.eecs.umich.edu/mibench/



Zhuoran Zhao received the B.S. in electrical engineering from Zhejiang University, Zhejiang, China in 2012, and the M.S. degree in electrical and computer engineering from The University of Texas at Austin, Austin, TX, in 2015, where he is currently working toward the Ph.D. degree. His current research interests include source-level simulation for software/hardware codesign, performance modeling of distributed embedded systems and fast prototyping of learning-based IoT applications.

Andreas Gerstlauer received his Ph.D. degree in Information and Computer Science from the University of California, Irvine (UCI) in 2004. He is currently an Associate Professor in the Electrical and Computer Engineering Department at The University of Texas at Austin. Prior to joining UT Austin, he was an Assistant Researcher with the Center for Embedded Computer Systems at UCI. He is co-author on 3 books and more than 80 refereed conference and journal publications. His research interests include system-level design automation, system modeling, design languages and methodologies, and embedded hardware and software synthesis. He is a senior member of the IEEE.

Lizy K. John received the PhD degree in computer engineering from the Pennsylvania State University in 1993. She currently holds the B.N. Gafford Professorship in the Electrical and Computer Engineering Department at The University of Texas at Austin. Professor John holds 10 U.S. patents and has published 16 book chapters, 180 refereed journal and conference publications, and approximately 60 workshop papers. She has coauthored books on Digital Systems Design using VHDL (Cengage Publishers), Digital Systems Design using Verilog (Cengage Publishers) and has edited a book on Computer Performance Evaluation and Benchmarking (CRC Press). She has also edited three books on workload characterization. Her research interests include microprocessor architecture, performance and power modeling, workload characterization, and low power architecture. She is a Fellow of the IEEE.