Host-Compiled Reliability Modeling for Fast Estimation of Architectural Vulnerabilities

Zhuoran Zhao, Dongwook Lee, Andreas Gerstlauer and Lizy Kurian John
Department of Electrical and Computer Engineering
The University of Texas at Austin
{zhuoran,dongwook.lee,gerstl,ljohn}@utexas.edu

***Abstract*—Due to ever increasing design complexities and continued technology scaling, reliability concerns due to single event upsets have become a key metric in electronic system design. During early design stages, soft error rates are usually modeled by estimating the Architectural Vulnerability Factor (AVF), which can be obtained by identifying the occupancy of Architecturally Correct Execution (ACE) bits. Previous approaches for AVF computation either rely on time-consuming cycle-accurate simulation of detailed micro-architectural structures or linear regression and other machine learning based estimation. However, accurate AVF estimation methodologies that can both capture the dynamic execution behavior and achieve fast simulation speed are lacking.**

In this paper, we propose a novel host-compiled simulation approach for AVF estimation with an emphasis on data storage structures. Our flow utilizes the retargetable back annotation approach to automatically annotate source-level simulation code with basic block reliability metrics. Based on producer-consumer dependency pair analysis, target architectural occupancy is simulated on the host and hardware components accesses are captured online to dynamically estimate AVF based on the occupancy statistics of micro-architectural structures.

Results of applying our model for estimation register file AVF on PowerPC targets running various benchmark suites show that an AVF estimation accuracy of more than 96% can be achieved while running simulations at close to source-level speeds in excess of 700 MIPS.

I. Introduction

With continued shrinking of feature sizes, reliability is quickly becoming one of the main issues in the design and operation of both embedded as well as general-purpose computer systems. Reliability of electronic systems against recoverable failures can be described using mean-time-between-failure (MTBF) or failures-in-time (FIT). The accurate estimation of these metrics is, however, challenging. By definition, this requires detailed and exhaustive long-term testing or complex fault injection experiments, which are often not feasible for architectural design space exploration, especially at early design stages.

In practice, designers and researchers use vulnerability metrics to predict and evaluate a certain design's reliability against soft errors from a probability-based perspective. In general, a vulnerability factor indicates the probability that an internal fault in a device structure will result in an externally visible fault [1]. Nevertheless, the estimation of such metrics still requires intensive, detailed simulation and profiling in order to accurately track micro-architectural states. Many previous efforts have been aimed at amortizing time-consuming cycle-accurate simulations for exploration across a wider range of the design space. Most approaches feed selected execution information into analytical, machine learning or regression based models for further prediction and estimation.

With ever evolving software content and growing dynamism and complexity of hardware systems, however, comprehensive simulations remain the primary tool for vulnerability estimation. At the same time, detailed mirco-architectural simulations quickly become prohibitive for all but the simplest architectures or workloads. As such, there is tremendous need for fast yet accurate estimation of architectural vulnerabilities at early design stages.

In this paper, we propose a novel approach for abstract, high-level reliability modeling using host-compiled simulation. In this approach, soft error vulnerability estimation is performed directly at the source level. In doing so, source code is back-annotated with information obtained from micro-architecture profiling to accurately track architectural occupancy and vulnerability. Back-annotated simulation models can in turn be easily integrated into transaction-level modeling (TLM) backplanes for co-simulation with other system components in order to provide a fast, reliability-aware prototyping infrastructure.

The rest of the paper is organized as follows: after an overview of background and related work in the following sections, details of the host-compiled estimation methodology will be described in Section III. Section IV then discusses the results of our experiments, and Section V presents the summary and conclusions.

A. Architectural Vulnerability Factor

The Architectural Vulnerability Factor (AVF) was introduced as a metric for measuring architecture-level reliability [1]. Mukherjee et al. define AVF as the probability of an error occurring in particular architecture components 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.

Two extreme examples are branch predictors and program counters, with AVFs of 0% and 100%, respectively. Most other structures will have an AVF that is in between these two extremes. Typically, data storage structures, such as register

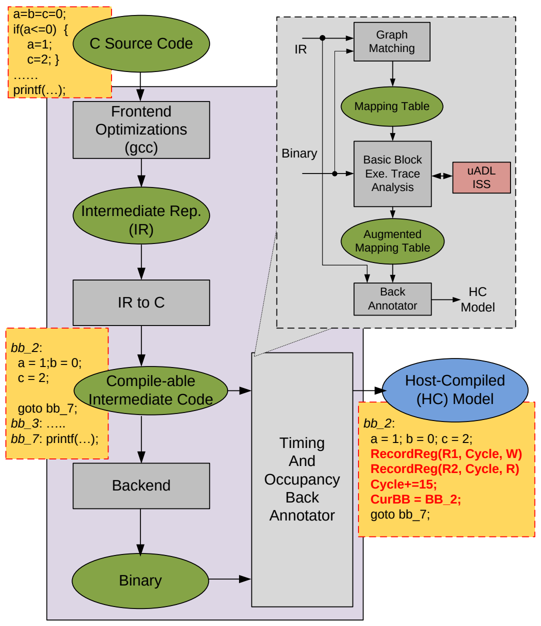


Fig. 1. Retargetable back-annotation for source-level simulation [8].

files and caches, will dominate usage and area occupancy. As such, they often contribute the most to the overall on-chip AVF. Impacted by both software and hardware, the AVF of data storage structures is therefore one of the key metrics for early assessment of soft error reliability.

B. Host-Compiled Modeling

Our work is focused on providing a light-weighted simulation infrastructure for AVF profiling and estimation with emphasis on data storage component vulnerability. Based on the observation that for any application binary, the corresponding architectural occupancy behavior can be replicated at the source code level, we adapt and extend existing techniques from the host-compiled simulation domain as a basis for developing our fast and accurate AVF estimation framework.

Host-compiled simulation based on native, source-level software execution has recently been introduced as an alternative to traditional instruction-set simulation (ISS). 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 [2]. Target execution information is added by prior back-annotation of the source with estimated target metrics [3], [4], [5], [6], [7]. In complete host-compiled models, annotated source code is then further wrapped into models of operating systems and processors that integrate into standard TLM backplanes. However, none of the existing host-compiled or source-level simulation approaches has considered reliability thus far.

The AVF simulation framework in this paper is based on the Retargetable Back Annotator (RBA) framework from [8] (Fig. 1). In RBA, application C code is passed through a generic cross-compiler front end to produce an intermediate representation (IR), which is then further translated back into compileable 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. The IR inherently provides a close representation of the final control data flow graph (CDFG) of the target code and hence is able to accurately reflect all data-dependent execution behavior. A mapping between IR and binary is established by control flow analysis of both CDFGs. Each basic block is characterized on a cycle-accurate reference simulator to extract timing information. Blocks are characterized in connection with all their possible immediate predecessors to accurately account for pipeline and other path-dependent intra-block effects. During back annotation, the IR's CDFG is then further augmented with previously characterized, path-dependent timing information. Finally, the IR code is compiled and simulated on the host to accumulate overall timing results.

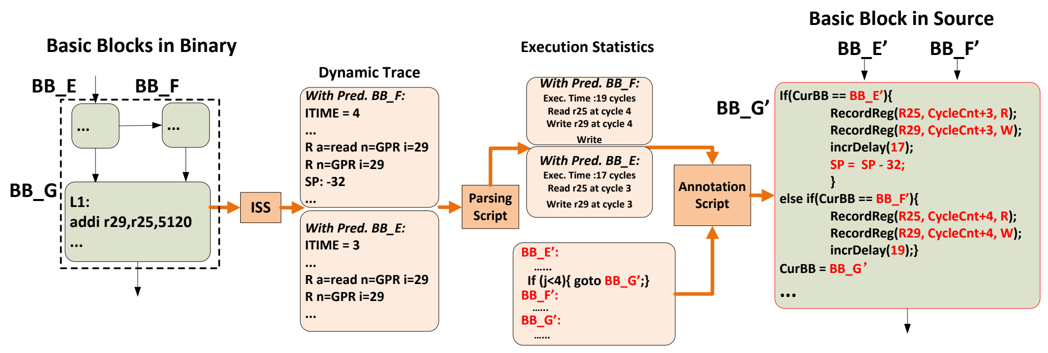
In this paper, we extend the existing back-annotation flow to account for accurate estimation of architectural vulnerability metrics. This is achieved by annotating the IR code with register access information that is obtained from pair-wise basic block characterizations on the reference simulator. Dynamic register access traces and target memory access traces are then generated in host simulation, and occupancy and AVF of the register file and data cache are further calculated.

II. Related Work

Many previous efforts has been focused on architecture-level reliability modeling and estimation. Li et al. propose an error injection based model to enable architecture level analysis of soft errors and vulnerability estimations in modern processors [9]. Programs are running on top of a probabilistic error generation and propagation model. However, the need for time-consuming simulation and observation limit the practical usefulness of such approaches.

Mukherjee et al. proposed a systematic approach to evaluate the AVF without actually injecting any errors [1]. The key idea of AVF estimation is to track the subset of processor state required for Architecturally Correct Execution (ACE). For example, the AVF for a single-bit storage cell is simply the fraction of time that it holds ACE bits. Assuming that all cells have equal raw fault rates, the AVF for a structure is the average AVF of its storage cells, or the average fraction of its cells that hold ACE bits at any point in time. Based on this analysis methodology, the evaluation of architecture level reliability can be converted into the analysis of AVF and further into instrumentation and profiling tasks to distinguish ACE and un-ACE bits. However, time consuming micro-architecture simulation and profiling is still needed, making it hard to integrate this methodology into iterative design space exploration across multiple architecture candidates and a wide range of benchmarks.

Many approaches have been proposed to improve on these early methods. Fu et al. characterize program vulnerability to soft errors in a high-performance out-of-order execution superscalar processor [10]. Their model includes instruction



```
...
L1:
addi r29,r25,5120
...
```

```
With Pred. BB_F:
ITIME = 4
...
R a=read n=GPR i=29
R n=GPR i=29
SP: -32
...
```

```
With Pred. BB_E:
ITIME = 3
...
R a=read n=GPR i=29
R n=GPR i=29
...
```

```
With Pred. BB_F:
Exec. Time :19 cycles
Read r25 at cycle 4
Write r29 at cycle 4
Write
```

```
With Pred. BB_E:
Exec. Time :17 cycles
Read r25 at cycle 3
Write r29 at cycle 3
```

```
BB_E':
  ......
 If (j<4){ goto BB_G';}
BB_F':
  ......
BB_G':
  ......
```

```
If(CurBB == BB_E'){
        RecordReg(R25, CycleCnt+3, R);
        RecordReg(R29, CycleCnt+3, W);
        incrDelay(17);
        SP =  SP - 32;
        }
else if(CurBB == BB_F'){
        RecordReg(R25, CycleCnt+4, R);
        RecordReg(R29, CycleCnt+4, W);
        incrDelay(19);}
CurBB = BB_G'
...
```

Fig. 2. Register accesses annotation.

window, reorder buffer, function units and wakeup table structures, all of which will significantly contribute to the overall architecture vulnerability. Their results show that reliability is a function of the interactions between the application and the hardware system, which is an important consideration for architecture researchers. Suh et al. propose a reliability-aware sampling-based approach called PHYS to reduce the memory access simulation time for cache AVF estimation [11]. To reduce profiling requirements and simulation time, other efforts decompose the AVF into different components, such as Program Vulnerability Factor (PVF) and Hardware Vulnerability Factor (HVF), where the latter can be obtained through static analysis of the hardware structures [12].

Although certain acceleration strategies are employed, cycle-accurate simulations across the entire program execution can not be avoided in the approaches above. As such, the speed-up is largely limited. Machine learning or regression based approaches have been proposed to deal with the simulation time problem. Based on observed correlations between performance metrics and AVF, Duan et al. proposed a machine learning approach to estimate the AVF and predict the reliability behavior [13]. It uses Boosted Regression Trees, a nonparametric, tree-based predictive modeling scheme, to identify the correlation between the AVF of key processor structures and various performance metrics. As an alternative, linear regression has been used to capture the relationship between other execution statistics and AVF for estimation of runtime AVF variations [14]. Furthermore, analytical models have been proposed to predict the AVF of a structure, in the first order, using statistics collected from relatively inexpensive profiling [15]. Finally, based on interval analysis [16], executions can be divided into individual windows for instrumentation and profiling, where AVF is calculated using a first order analytical model. In all cases, however, machine learning or analytical approaches 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 adapting and extending source-level simulation concepts to the problem of reliability estimation. The architectural behavior of the entire execution is thereby natively simulated on the host, leveraging fast source-level software execution and high-level processor model simulation. Back-annotation of profiled target characteristics ensures estimation accuracy, while host simulation accurately captures all dynamic application/architecture interactions.

III. Host Compiled AVF Modeling

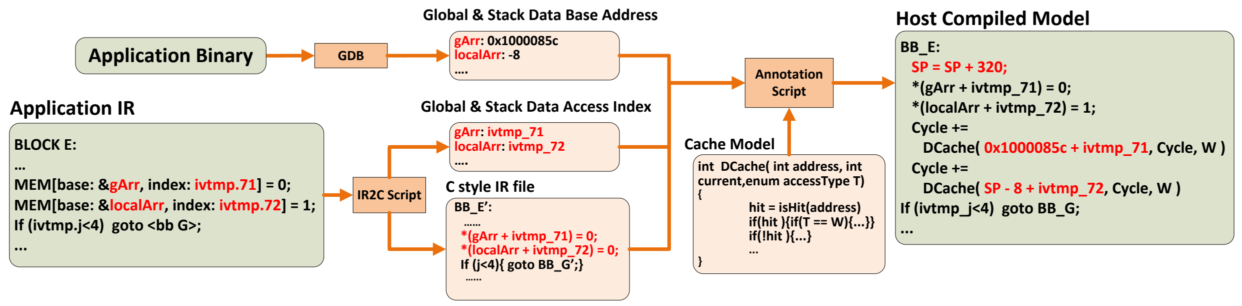
The key idea in host-compiled modeling is to compile and execute application code natively on the host with back-annotated execution metrics obtained from target profiling. In our work, the retargetable back-annotation process is extended with a producer-consumer analysis to extract architectural occupancy information during target profiling. In the host-compiled model, target memory traces are regenerated based on memory access information from debugger and application intermediate representation (IR) and feed into a light-weight cache model. With the back-annotated occupancy information and reconstructed memory trace, the host-compiled simulation code can duplicate the register file and data cache access behavior of the target code and further provide target execution tracing for dynamic AVF estimation.

The back-annotation flow for host-compiled AVF modeling is shown in Fig. 2. During the profiling stage, the timing of architectural activities, such as register accesses and stack pointer changes are recorded for each basic block. Based on the mapping table between the CDFG of the IR and the binary, the IR code is then back-annotated with the corresponding activity information. The memory trace reconstruction flow is shown in Fig. 3. The base address information of global and stack data from debugger and access index from IR are combined and annotated into the C style IR code for the memory trace reconstruction. The cache access outcome is simulated online by a cache model and a fixed performance penalty will be added in case of a cache miss.

The process in Fig. 2 and Fig. 3 is automated and introduces negligible overhead that is amortized across the duration of long-running simulations.

A. Basic Block Characterization

During the profiling stage, basic blocks are executed in pairwise combination with all their possible immediate predecessors. This pair-wise characterization corresponds to an



```
gArr: 0x1000085c
localArr: -8
....
```

```
BLOCK E:
...
MEM[base: &gArr, index: ivtmp.71] = 0;
MEM[base: &localArr, index: ivtmp.72] = 1;
If (ivtmp.j<4)  goto <bb G>;
...
```

```
gArr: ivtmp_71
localArr: ivtmp_72
....
```

```
BB_E':
  ......
  *(gArr + ivtmp_71) = 0;
  *(localArr + ivtmp_72) = 0;
  If (j<4){ goto BB_G';}
  ......
```

```
int  DCache( int address, int
current,enum accessType T)
{
          hit = isHit(address)
          if(hit ){if(T == W){...}}
          if(!hit ){...}
          ...
}
```

```
BB_E:
  SP = SP + 320;
  *(gArr + ivtmp_71) = 0;
  *(localArr + ivtmp_72) = 1;
  Cycle +=
    DCache( 0x1000085c + ivtmp_71, Cycle, W )
  Cycle +=
    DCache( SP - 8 + ivtmp_72, Cycle, W )
If (ivtmp_j<4)  goto BB_G;
...
```

Fig. 3. Memory access trace reconstruction.

edge profiling of the CDFG and is done to collect execution metrics across all possible execution contexts, which provides a trade-off between profiling cost and accuracy compared to full path or standalone node profiling. Each collected block pair is executed on the cycle-accurate reference simulator to collect traces of various execution events. Note that during the profiling, all cache lines are initialized to be valid and every cache access is assumed to be a cache hit. The real cache outcome will be calculated in the host-compiled model and miss penalty will be added online as a fixed number of cycles.

In our case, for AVF modeling, collected execution metrics mainly include the cycle duration of the basic block under characterization, the time stamp offsets of data structure accesses relative to the start of the current block and the change of stack pointer ΔSP. The duration of each block will later be used to accumulate the overall execution time, whereas the actual register access times will be calculated online during simulation as the sum of the accumulated execution time at the entry of the current block plus the characterized access offset. The ΔSP will also be accumulated to track the value of stack pointer in order to calculate the base address of local variable on stack.

As shown in the example of Fig. 2, 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 the host execution. For example, if in a particular simulated execution of block BB_G the previously executed block was BB'_F, which corresponds to block BB_F in the binary, then the profiling metrics for predecessor BB_F are picked, and the access time for $R29$ is estimated as *3+Current Execution Cycle*, while the access type is a *register write*.

B. Memory Trace Reconstruction

The pairwise basic block characterization can only capture limited architectural behaviors. The memory access trace, which is essential for the cache AVF estimation, still require the entire program execution. Instead of execute the target binary on a cycle accurate simulator, we reconstruct the memory trace on source-level with annotation of memory information from debugger and application IR. The target memory access trace is then generated online in the host-compiled model and feed into a cache model for the estimation of the target data cache AVF.

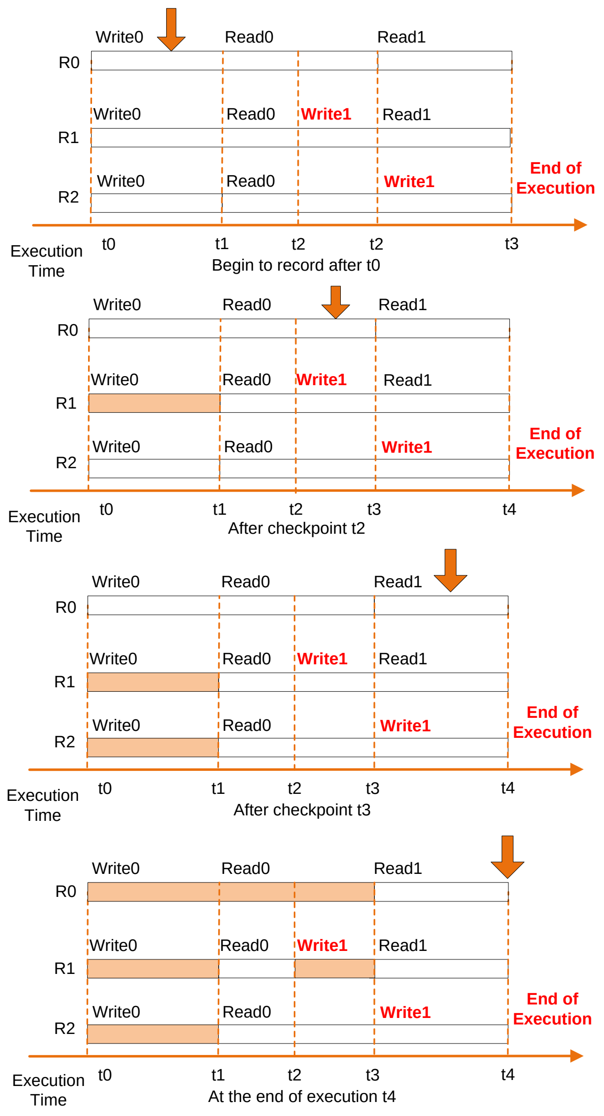


Fig. 4. Producer and Consumer Pair Analysis.

Here we only consider the memory access to the global data and stack data. Such simplification is based on the observation that many embedded applications only implement these two memory mechanisms in order to minimize the resources utilization and power consumption.

1) Memory Access on Global Data: In order to reconstruct the memory access of global data, their base addresses access index are typically needed. The base addresses of static global data are fixed address numbers and stored in the symbol table of ELF file. We apply the original application binary (ELF) to the GDB to obtain these information. The access index however can be dynamically changed during execution and is available in the form of C variables or expressions the source file. In our case, we leverage our Retargetable Back Annotator to parse and extract those index information during the IR to C conversion stage. As the example shown in Fig. 3, the base address of global array *gArr* is *0x1000085c*. The access index of *gArr*, *ivtmp_71*, can be captured in the application IR file and translated to C style syntax by the IR2C script. Finally such information is combined and annotated to the C style IR file. During the execution, the base address of *gArr* is a constant as annotated, and the dynamic changes of *ivtmp_71* is calculated in the host-compiled model to reconstruct the access trace.

2) Memory Access on Stack Data: Different from the global data, the stack data access is more complicated since both of their base addresses and access index can be changed dynamically. Previous work relying on parsing original C code does not clearly resolve this problem [17]. The stack allocation decision can be only decided at the backend optimization stage during compilation. Thus one can not rely on the application's IR or C source code to deduce the stack layout. Besides, even the stack frame layout of each function can be decided, the value of the stack pointer is changing dynamically during the execution.

To track the value of the stack pointer, we create a recorder variable *SP* in the source code. The change of stack pointer, Δ*SP*, is recorded in the basic block characterization and annotated to the host compiled model. The host compiled model will then accumulate such changes so that inside each basic block we can know the current target stack pointer value.

In order to know the base addresses of the local variables in current stack frame, we rely on the GDB and use command *info address symbol* to get their base address offset related with the current stack pointer. So for stack data the annotated access address should be *SP + offset + index*. In the example shown in Fig. 3, the base address of *localArr* is *SP - 8*, index is *ivtmp_71*, so the annotated value should be *SP - 8 + ivtmp_71*.

C. Occupancy Analysis

A soft error in a data storage structure will manifest itself when faulty data is read out of a particular location. Thus, those structures' AVF can be obtained by estimating the occupancy of all the variables waiting to be consumed, and the estimation of AVF can be converted into the problem of capturing the variable life times of all resident data. In order to estimate the AVF for data storage structures, we apply a producer-consumer analysis to dynamically capture such variable live times during simulation.

1) Register File Analysis: Taking the register file as an example, 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 length of the total program execution. 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. 4. The key idea in this analysis is to capture the time stamp at the end of each variable's life time, 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 execution. In both cases, the existing data in the access location is guaranteed to not be consumed again and the variable life time can be calculated. In this way, the architectural occupancy of a certain storage structure is divided into atomic variable life time periods defined by these check points.

An example of the producer-consumer analysis process is shown in Fig. 4. The execution begins with initial write operations to registers $R0, R1$ and $R2$. During the execution, $R1$ and $R2$ are then written again at times $t2$ and $t3$, respectively, which are highlighted in red as the check points for both registers. Finally, time $t4$ as the end of the execution will be the checkpoint for all registers. During the simulation, the access times of all the latest write and read operations are continously recorded and updated. For each storage unit, whenever a checkpoint is reached, the time difference between the latest reading and writing will be calculated and accumulated as the occupancy time. After the final checkpoint at the end of simulation, all the storage cells' occupancy time can be obtained and later used for AVF estimation.

In Fig. 4, the start time $t0$ is first recorded as the latest write operation time stamp for all registers. At $t1$, the latest read operation time for all registers is recorded as $t1$. Then, at $t2$, $R1$ reaches its check point, where the difference between $t1$ and $t0$ is accumulated as $R1$'s occupancy time, represented as shaded bar in Fig. 4. At $t3$, $R2$ 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 the registers hit their checkpoint. The period between $t0$ and $t3$ is accumulated as $R0's$ occupancy time, and the period between $t2$ and $t3$ is added to $R1's$ occupancy time. For $R2$, since there is no subsequent read operation after $t3$, the write operation at $t3$ is discarded. In this way, at the end of the execution, the occupancy periods of all registers are captured as shown by the shaded bars in the figure.

2) Data Cache Analysis: Different from register file, the estimation of data cache can be non-trivial. In the case of register, the beginning of a variable's life is simply a register write, so the checkpoints only include register write and the execution end. For cache operations, there are four different

```
//Global variable to record register accesses
unsigned long  RegFile[32][3] = {{0,0,0},……,{0,0,0}};


//Accumulate occupancy time based on prod.-cons. analysis
void RecordReg( int regID,
                long accessCycle,
                int operation)
{
  if(operation == W) {//Check whether it is checkpoint
        if (RegFile[regID][read]>RegFile[regID][ write])
              RegFile[regID][Acc]+=   //Accumulate occupancy
                   (RegFile[regID][read] - RegFile[regID][write]);
        RegFile[regID][read] = 0;
        RegFile[regID][write] = accessCycle; }
  if(operation == R) //If it is read operation, then update recorder
              RegFile[regID][read] = accessCycle;
}


//Collect the occupancy duration at the end of execution
void FinalRegFileOccp()
{
              int i;
              for (i=0; i< 32; i++)//Checkpoint for all registers
              if (RegFile[regID][read]>RegFile[regID][ write])
              RegFile[regID][Acc]+= //Accumulate occupancy
                 (RegFile[regID][read] - RegFile[regID][write]);
}
…
```

Fig. 5. Host-compiled AVF model for register file.

scenarios from the processor's perspective which can be *write miss, write hit, read miss* and *read hit*. Besides the write operations, a cache miss on read can also mark the beginning of a cache data life since a new cache line will be filled into the cache.

So the checkpoints for the producer-consumer analysis of data cache include *write operations on both cache miss and hit*, *read operations on cache miss* and *the end of a program execution*. Upon such events, the occupancy time can be accumulated since the cache resident data in the access location is either overwritten or evicted, and the old data is guaranteed to not be consumed again. After identifying the checkpoints, all the other analysis method of data cache is same with register file.

D. AVF Estimation

1) Register File AVF Estimation: Sample code for implementation of the producer-consumer analysis of a register file within an overall host-compiled simulation framework is shown in Fig. 5. A global *RegFile* array is introduced to record 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. The function *RecordReg* is inserted into each basic block based on the back annotation information as shown in Fig. 2. During host-compiled simulation, *RecordReg* will be called to monitor and record each register access point. Internally, the function applies a producer-consumer analysis method to update the corresponding access and occupancy time information in the global *RegFile* array. Finally, at the end of the execution and hence at the final checkpoint, a global *FinalRegFileOccp*

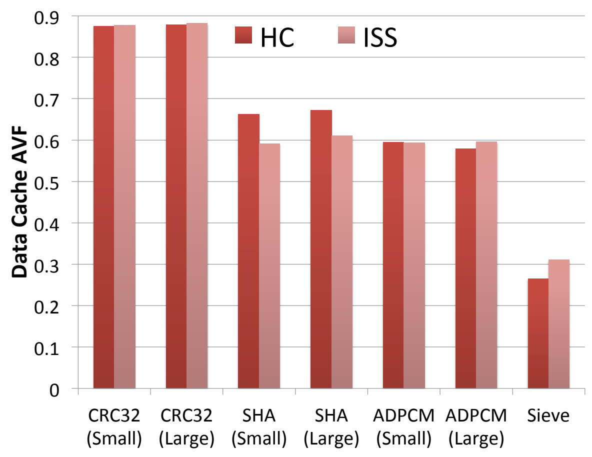


Fig. 6. Comparison of Cache AVF estimation for PowerPC Z4 target.

will be invoked to collect all the remaining occupancy times for every register.Note that the register IDs and cycle offsets are obtained through pair-wise characterization as previously described in Section III. In order to further obtain the absolute access times, we also need to compute the accumulated execution time at the beginning of a block's execution. We therefore run the register accesses recorder along with a host-compiled timing model, such that the producer-consumer analysis can accurately take into account the global execution times. The *accessCycle* for each operation is thereby calculated as the sum of accumulated execution cycles at the entry of the current basic block and the cycle offset of a particular access as originally characterized relative to the beginning of the current basic block.

2) Data Cache AVF Estimation: For the implementation of the producer-consumer analysis of data cache, similarly, a global *DCache* array is used to record the latest line-fill/overwrite time, latest read on a cache hit time and the total accumulated occupancy time for each word in the data cache. During the execution, the memory trace generated from host-compiled model is feed into a cache access recorder function. As discussed before, the checkpoints in a cache analysis are also dependent on the hit and miss information. So before perform the producer-consumer analysis, we need to invoke the cache model to decide cache access outcome, and upon cache write and cache read miss, i.e. checkpoints, the occupancy time will be accumulated. Otherwise if a cache access is a read hit, the latest read recorder will be updated.

Based on the fact that the dynamic execution trace of a program will usually contain many more executions of a basic block than there are block instances in the static CDFG, our approach of profiling each basic block through a one-time effort and instead performing the producer and consumer analysis during source-level execution can help to dramatically reduce the required simulation time for reliability estimation.

IV. Experimental Results

We have implemented our automated, host-compiled AVF model generation flow in Python using Freescale's retargetable

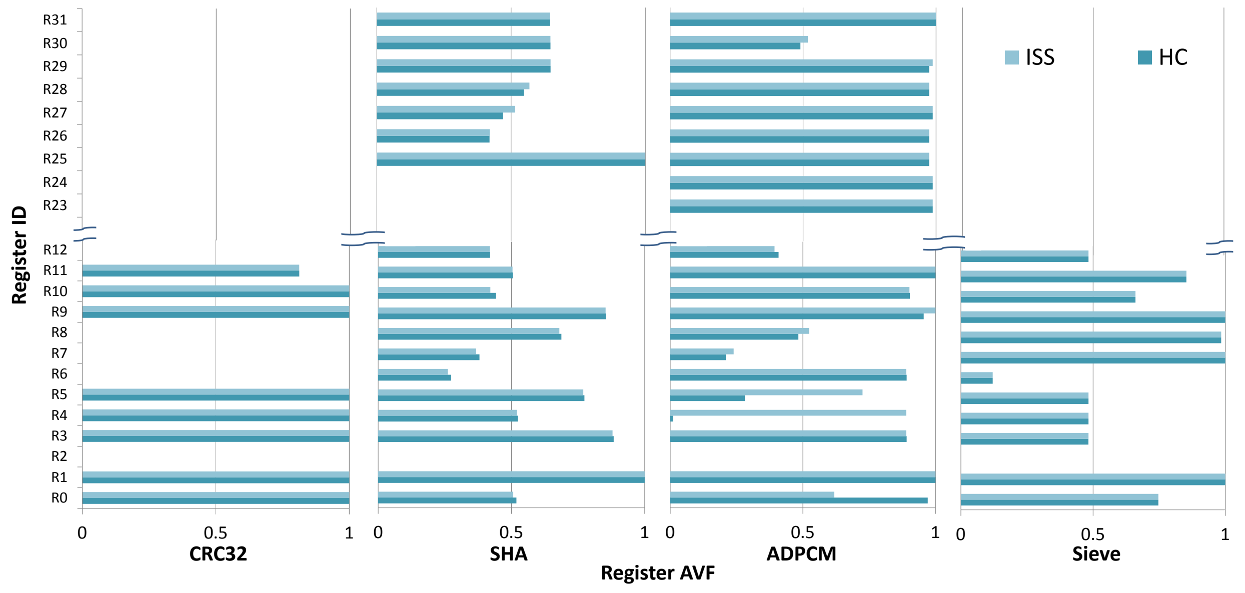


Fig. 8. Occupancy of each individual register for Z4 target and small input data set.

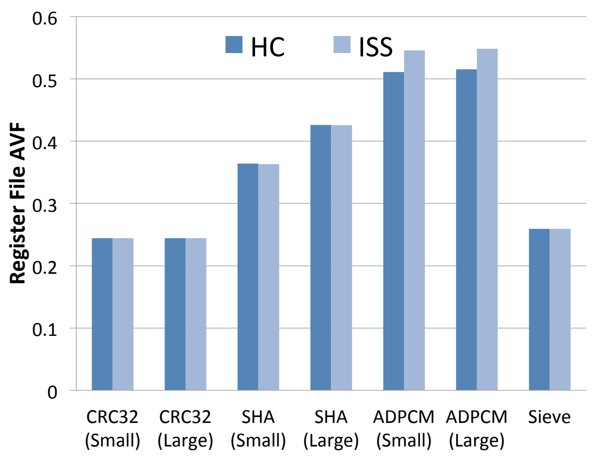


Fig. 7. Comparison of Register File AVF estimation for PowerPC Z4 target.

uADL ISS framework as cycle-accurate timing and micro-architectural behavior reference [18]. To demonstrate the benefits of our flow, we applied it to several standard benchmarks running on a set of generic PowerPC based targets. All AVF model generation and simulation was performed on a quad-core Intel i7 workstation running at 2.6 GHz.

We selected three benchmarks (CRC32, SHA, and ADPCM) from the MiBench suite with both small and large data sets as well as a custom application (Eratosthenes Sieve) for validating our flow [19]. The latter calculates prime numbers in the range 0 to 500,000.

The target we evaluated is an in-order, e200 z4-like dual-issue core with 16KB 4-way associative L1 data cache and instruction cache. The target does not include floating point unit or dynamic branch predictor. A gcc-4.4.5 cross-compiler was used to generate code for both targets and provide debugging information.

A. Register File and Data Cache Occupancy

By monitoring the producer and consumer pairs, we simulated the occupancy of each register throughout the entire execution time of each application, and we calculated the register file AVF as the average occupancy of each register.

Fig. 7 compare the register file occupancy results obtained from the host-compiled (HC) models against cycle-accurate simulations of complete application runs executed on the corresponding ISS reference models. The average occupancy error is less than 0.98% The largest error of around 3.4% occurred in the ADPCM benchmark. For all other benchmarks, the estimation error is less than 0.1% on both targets. The ADPCM benchmark exhibits relatively larger errors since its CDFG 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. Corresponding timing inaccuracies manifest themselves as jitter in recorded register access times, which in turn lead to errors in AVF estimations.

Fig. 6 compare the data cache occupancy results obtained from the HC models against cycle-accurate simulations reference models. The average occupancy error is less than 0.98% The largest error of around 3.4% occurred in the SHA benchmark. For all other benchmarks, the estimation error is less than 0.1% on both targets.

As expected, the occupancy and AVF data storage structures do not vary significantly with the size of the data set.

Fig. 8 breaks the overall AVF of the whole register file into AVF metrics for each individual register. Registers $R13$ to $R22$ are reserved and the occupancy is always 0%. As such, they are omitted in Fig. 8. In the PowerPC calling convention, registers $R23$ to $R31$ are usually used as link registers to store return addresses. Hence, they will only be occupied when there are

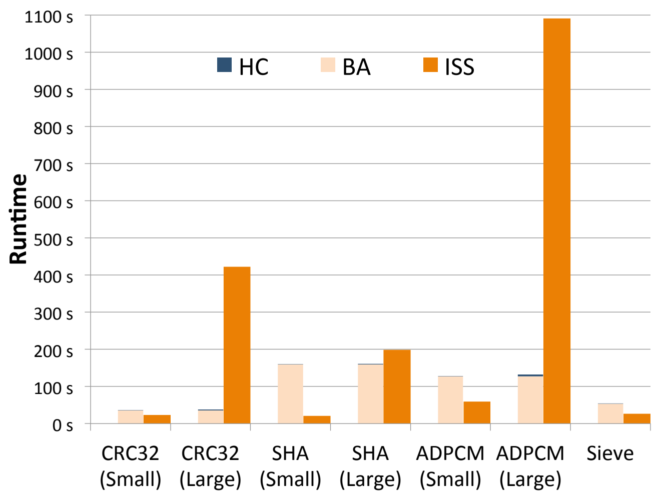


Fig. 9. Total runtime comparison.

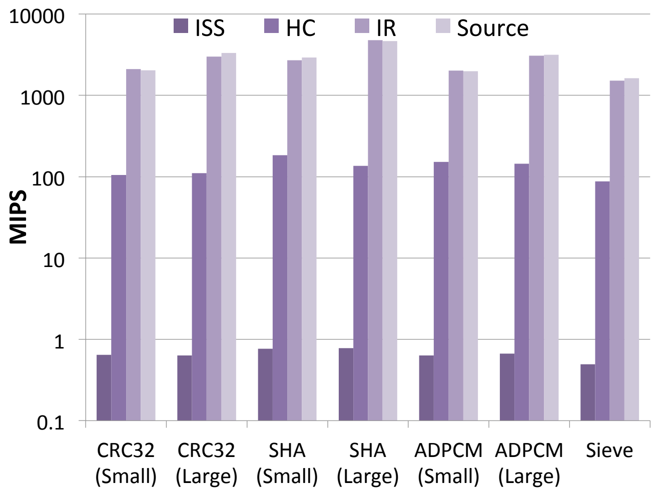


Fig. 10. Simulation speed comparison.

a number of function calls during the execution, which is the case in the ADPCM and SHA benchmarks. Other benchmarks consist of a single function only and link registers remain unused. Results show that the host-compiled simulation can accurately replicate the dynamic occupancy of each register with less than 1% error on average. This shows the ability of our approach to enable detailed analysis at fine structural granularity.

B. Simulation Speed

We compared the simulation speed and runtime of our approach against a reference ISS flow. Note that while running the reference ISS, we only recorded the runtime without the overhead of collecting and parsing of dynamically generated traces, which is typically the most time-consuming bottleneck of vulnerability analysis in a traditional flow.

Fig. 9 shows that for small data sets and short simulation runs, the benefits of faster host-compiled simulations do not outweigh the overhead introduced by the extra back-annotation (BA) process. However, as input data sets and simulation lengths grow, combined back-annotation and host-compiled simulation runtimes increase at a much slower rate than the total ISS time. Furthermore, back-annotation is a one-time effort. Once generated, resulting host-compiled models can be repeatedly resimulated under varying input scenarios. Note that runtimes are dominated by back-annotation time, and HC simulation time contributes less than 1%. The relatively large runtime for SHA is due to its large number of basic blocks. As a result, there is a relatively larger overhead introduced by the back-annotation process.

Finally, Fig. 10 shows the simulation throughput of back-annotated host-compiled models as compared to pure source- or IR-level simulations as well as a traditional cycle-accurate ISS execution. An average of 770 MIPS can be achieved by our host-compiled simulation model. This is several orders of magnitude faster than an equivalent ISS execution, and only an average of 3.5 times slower than a native execution of the unmodified application source code.

V. Summary and Conclusions

In this paper, we proposed a host-compiled architectural vulnerability factor modeling methodology. Based on a retargetable back-annotation for basic block characterization and profiling, our model is able to simulate the micro-architectural occupancy behavior directly at source level. Applying an online producer-consumer analysis for the simulated access traces, our model is able to accurately duplicate the dynamic access patterns of storage structures. Our approach can be easily integrated into traditional TLM-based virtual platform prototypes to provide fast system reliability feedback for HW/SW co-design and early design space exploration. Applied to several standard benchmarks, the host-compiled AVF modeling performance can reach up to 870 MIPS with more than 96% accuracy compared to a traditional ISS-based vulnerability estimation method.

Future work will include extending the approach to reliability modeling of other (micro-) architectural structures, including support for more advanced out-of-order architectures and additional benchmarks.

References

[1] 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.

[2] A. Gerstlauer, "Host-compiled simulation of multi-core platforms," in *International Symposium on Rapid System Prototyping (RSP)*, Jun. 2010.

[3] 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.

[4] A. Bouchhima *et al.*, "Automatic instrumentation of embedded software for high levelhardware/software co-simulation," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009.

[5] S. Stattelmann *et al.*, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *Design Automation Conference (DAC)*, 2011.

[6] Z. Wang *et al.*, "Accurate source-level simulation of embedded software with respect to compiler optimizations," in *Design, Automation and Test in Europe (DATE)*, 2012.

[7] 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.

[8] 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.

[9] X. Li, S. Adve, P. Bose, and J. Rivers, “Softarch: an architecture-level tool for modeling and analyzing soft errors,” in *International Conference on Dependable Systems and Networks (DSN)*, 2005.

[10] 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.

[11] J. Suh, M. Annavaram, and M. Dubois, “Phys: Profiled-hybrid sampling for soft error reliability benchmarking,” in *International Conference on Dependable Systems and Networks (DSN)*, June 2013.

[12] V. Sridharan and D. R. Kaeli, “Using hardware vulnerability factors to enhance avf analysis,” in *International Symposium on Computer Architecture (ISCA)*, 2010.

[13] 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.

[14] 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.

[15] 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.

[16] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, “A mechanistic performance model for superscalar out-of-order processors,” *ACM Transactions on Computer Systems*, vol. 27, pp. 3:1–3:37, May 2009.

[17] 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.

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

[19] Mibench version 1.0. [Online]. Available: http://www.eecs.umich.edu/mibench/