Exploiting Instruction Reuse to Enhance Microprocessor Simulation

Ravi Bhargava, Lizy K. John, Francisco Matus [*]
Electrical and Computer Engineering Department
The University of Texas at Austin
{ravib,ljohn,matus}@ece.utexas.edu

Abstract

The use of software simulation to model modern high-performance microprocessors is becoming increasingly challenging as microprocessors grow in complexity. Accurate and meaningful performance analysis of an out-of-order, superscalar microprocessor is complicated by the fact that no component of the system is truly orthogonal to the rest of the system. At the same time, each component of the system requires a fine level of simulation. Therefore, there exists a tradeoff between the accuracy of results and the amount of time necessary to create a simulation environment and perform the simulations.

Based on the behavior of programs, this study proposes a structure that can decrease the simulation time of microprocessor software simulators, and in some cases even improve the accuracy of simulation. This is accomplished by taking advantage of the knowledge that the same static instructions are executed many times dynamically. We recreate an approximate copy of the object code, which we call the resurrected code, using instructions from the dynamic instruction stream of the simulator. For any style of simulation, the resurrected code can be used to decrease the time spent decoding instructions, which is often significant in simulation as it is in actual processor execution. Along with decreasing simulation time, the resurrected code provides an improvement in accuracy for trace based simulators which are not provided with a program code segment. In trace based simulation, it is possible to fetch instructions from the resurrected code structure after a mispredicted branch and then introduce them into the simulated processor as an actual processor would do. This allows for a more realistic modeling of mispredicted path execution. In addition, the structure provides an elegant method for gathering statistical information regarding the use of specific static instructions, and becomes an easy means for quickly specifying internal simulator-specific hints and directions.

In this paper, implementations of the resurrected code are detailed, as well as the treatment of wrong path instructions in trace-driven timing simulators. We describe and analyze the impact of introducing wrong path speculative instructions for a series of C, C++, Java, and Fortran programs in a trace-driven processor simulation environment. We find that for 92% to 99% of mispredicted branches, the resurrected code can supply all the proper wrong path instructions needed to more accurately model mispredicted paths.

[*] *L. John is supported by the National Science Foundation under Grants CCR-9796098 (CAREER Award), and EIA-9807112, and a grant from the Texas Advanced Technology Program. F. Matus is also with Advanced Micro Devices.*

1 Introduction

Accurate simulation of modern microprocessors is an important process in both academic and industrial research [1]. The time required to produce simulated results is often just as important as the accuracy of the simulation, if not more important. This time includes creating the simulation environment, validating the tools used in simulation, producing relevant test cases, and the running of the simulations themselves.

It is common practice to simulate the performance of microprocessors using combinations of software and hardware techniques. As modern processors are becoming more complex, the process of performing accurate, cycle-level simulation is becoming increasingly challenging. To maintain the level of accuracy needed for meaningful simulations, simulators are becoming more detailed, less portable, less flexible, and slower. While the accuracy of the results are important, obtaining them in reasonable time is equally desirable.

1.1 Methods of Simulation

Functional simulators [2, 3, 4] simulate the entire register level transfer of data and can reproduce identical program outputs to the processor that it is modeling. Functional simulators are often execution driven or program driven. Execution driven simulation [5] is a relatively fast technique that executes many of the instructions on the host machine instead of simulating all of the instructions. Execution-driven simulators take an executable (often cross-compiled into a simulated instruction set architecture) as the input. Program-driven simulation is similar to execution-driven simulation. Program-driven simulation [4] will consume an uninstrumented executable and perform analysis on this executable.

Functional simulators are very accurate tools for simulating microarchitectures. Combining this accuracy with public releases of validated functional simulators, functional simulation is becoming a preferred method of computer architecture performance evaluation. Unfortunately, this style of simulation is often restricted by the complexity of modeling functionality. It is often the case that functional simulators require source code and a modified compiler to produce an executable that can be simulated.

Sometimes, the source code is even compiled into its own unique instruction set architectures (ISA) [2].

Another restriction of functional simulation occurs when modeling the functionality of operating system calls. This often requires extensive coding, operating system specific tricks, or restricting the type of executables that can be modeled. Dealing with these issues of system calls results in simulators that either cannot execute all types of executables, or simulators that become tightly coupled with the operating system on which they are being run.

In general, functional, execution-driven simulation requires a large and complex infrastructure. This is very difficult for any architecture, but especially for complex CISC architectures like the X86. The effort to build the system is man-hour intensive, as is the verification process. Unless the infrastructure is correctly partitioned, it is quite difficult to make changes without causing many potential bugs.

Of course, current functional simulators are designed to run today's popular benchmarks which are often accompanied by source code, allowing for simulator-dependent alterations. Whether such benchmarks are representative of common and commercial applications is debatable, but it is not debatable that the modeling of state-of-the-art applications and workloads should be the force driving microprocessor design [1]. With this in mind, it is desirable to model applications that may not have publicly available source code. Even with source code, these applications tend to contain more optimization (including hand optimization) than popular compilers can achieve. Hence, we consider it extremely important to be able to accurately model these workloads.

Trace-driven timing simulators have the ability to simulate more quickly than functional simulators and tend to have more freedom in the development stage. In trace-driven simulation, details from the dynamic execution stream of a process or processes are recorded and then fed directly or via a file to a software timing model of a microprocessor. Traces are most commonly produced by software monitoring methods such as trapping, manipulating object code [6] or by hardware monitoring methods [7].

Trace generation tools such as Shade [8] are very robust. Shade can trace any single-threaded SPARC executable regardless of system calls, compiler, or availability of source code. In addition, it is possible to trace operating system effects without any dilation by a hardware trace generator [7]. For functional simulators to study these effects, instrumentation and modifications of the entire operating system calls are required. With these flexibilities, more realistic workloads can be analyzed using traces.

Another motivation for using trace-driven timing simulation is the current existence of many trace-driven simulators and tracing tools. Trace-driven simulation also enjoys the advantages of portability and flexibility. In its simplest form, it requires only a trace and simulation software. While trace generation systems have several constraints such as a dependency on operating system, compilers, etc., the generated traces and the simulator can be transported across platforms rather easily. For applications with no source code, traces are the most convenient way to capture the instruction stream of the program. Trace-driven simulation helps to decouple specific issues in trace generation from issues in simulation. Several microprocessor companies tend to favor separate teams working on trace generation tools and simulation. Due to stability and familiarity, trace based simulations supply a high level of comfort and confidence.

1.2 Inaccuracies in Trace Based Simulation

While developing and maintaining a trace based simulator is relatively easier than other alternatives, a primary drawback is the inability to accurately simulate speculative instruction fetching and the subsequent execution. Specifically, speculative instructions are not yet adequately represented in traces. In state-of-the-art processors, when a branch is predicted, the processor starts to fetch instructions from the predicted target address. Many of these instructions are decoded, issued, and even executed, but are not committed until the actual branch target is resolved. If there is a misprediction, these instructions are flushed (squashed) from the processor and are never seen by the tracing tool since tracing tools see only the executed instruction stream.

Placing all the proper data in a trace so that a simulator can accurately execute misspeculated instructions is a difficult and expensive task. One proposed remedy is to use a trace-driven simulator along with some other supporting techniques to acquire the information needed to properly simulate speculative instruction execution. One such method is employed by Reilly and Edmondson with the

Alpha Microprocessor simulator [9] in which they use AINT [10] in conjunction with a trace-driven simulator to supply basic blocks of instructions from speculated addresses. However, many of such platform-restricted mechanisms need access to an instrumented executable or symbol tables.

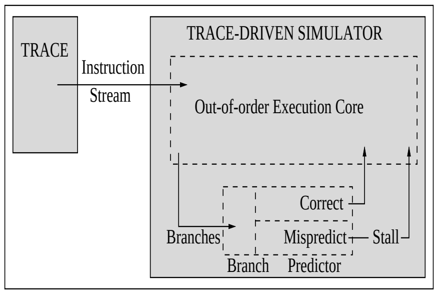
The process of squashing has often been modeled in trace-driven simulation by stalling instruction fetching until the mispredicted branch has been evaluated. Some simulators incorporate a fixed penalty to model the branch misprediction [11]. This ignores the fact that the wrong path speculative instructions consume processor resources and affect the future state of the processor. Moudgill et al quantified the impact of not simulating mispredicted paths on a four-issue processor using programs from the SPEC95 integer benchmark [12]. They found that the variation in instructions completed per cycle is small (in all but one case it is less than 0.5%). They also found that mispredicted memory references may lead to additional cache hits, acting as a natural type of prefetch mechanism. Although these results are encouraging for users of trace-driven simulators, the growing concern is that increasing instruction issue widths and degrees of speculation will lead to an unacceptable level of error.

In this paper, we propose a structure to improve the speed of many varieties of processor simulators by taking advantage of program behavior, specifically the frequent reuse of instructions within sections of a program and within the program as a whole. The simulator strategically stores information pertaining to each unique instruction that is decoded. This process creates an approximate copy of the object code, which we call the *resurrected code* [13]. When simulation takes place, dynamic instructions whose static information has already been determined can forgo most of the time-consuming decode stage. This also provides a means for simulator-specific internal message passing. In addition, the same structure can be used to improve the simulation accuracy of trace based simulators. The resurrected code become the source for fetching instructions along mispredicted paths. Mispredicted branch targets can access the resurrected code and instructions can be fetched from there.

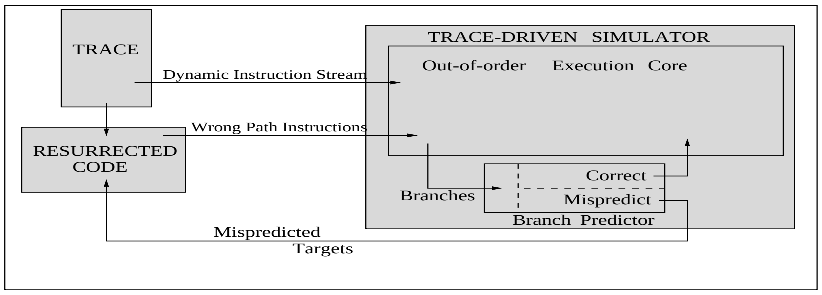
The paper is organized in the following manner. Section 2 discusses the implementation of the resurrected code structure. Section 3 describes the benefits, including decreased simulation time and improved accuracy of trace based simulation. Section 4 explains our simulation environment, tracing tools and benchmarks. Section 5 is an analysis of the impact of the resurrected code structure on trace-driven simulation. Section 6 concludes the paper.

2 Implementation of Resurrected Code

Figure 1.a illustrates the traditional approach to trace-driven simulation. In this approach, the simulator receives instructions sequentially from the trace and these instructions trigger the simulation. When a branch instruction is encountered, a mispredicted branch results in the simulator effectively stalling until the branch is resolved and then a uniform branch misprediction penalty is applied. The instruction streams along the mispredicted paths are not normally available for simulation. The resurrected code proposed in this paper becomes a source for fetching the instructions for the mispredicted paths as shown in Figure 1.b.



(a) Traditional



(b) With Resurrected Code

Figure 1: Traditional Use of Trace-Driven Simulator versus Trace-Driven Simulation with Resurrected Code

2.1 Data Structure

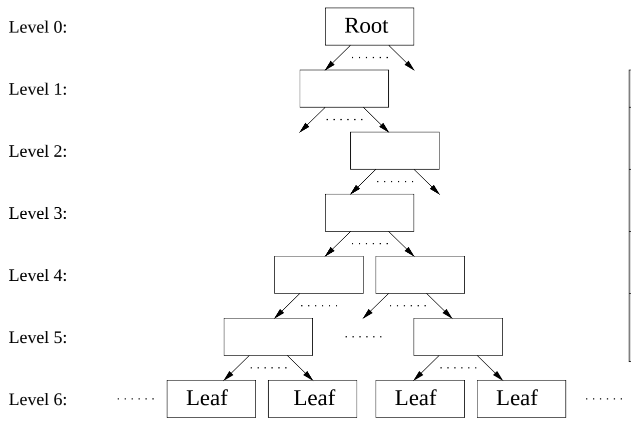
Ideally, the structure that holds the instruction information should be memory efficient and quickly accessible both while creating and using it. Instructions have been shown to have both temporal and spatial locality, so instructions should, in general, be blocked together. On the other hand, one in every four to six instructions can disrupt the control flow of the program. So it is quite possible that within these blocks there will be small “holes” where instructions are never reached during a certain execution of a program.

Accounting for the above observations, we implement a dynamic, tree-like structure that is directly indexable by the program counter (PC), and attempts to minimize the amount of memory allocated to the holes. This *resurrection tree* is composed of nodes, where each non-leaf node contains pointers (in C) to more nodes. Each non-leaf node need not contain any information other than the location of its children. Leaf nodes do not contain the array of node pointers, but instead contain information about the instructions they are representing as well as any additional information that suits the user.

It would be convenient for the nodes at each level of the tree to be represented by one structure and therefore have the same number of children. This would require that each level of the tree may potentially have $2^{n \cdot x}$ nodes, where n is the current level of the tree and x is the increase in size (in bits) between levels. For example, a byte-addressable 32-bit address space like that in the X86 architecture, could be represented by a four level tree (not including the root) where each node has $2^{8 \cdot n}$ potential children per non-leaf node or perhaps a six level tree where each level has $2^{6 \cdot n}$ potential children per non-leaf node. Figure 2 depicts a tree for a 32-bit address space with word (4 bytes) addressable instructions, like in the UltraSPARC and other RISC processors. Since the last two bits of the program counter are immaterial, the tree can be represented by six levels (seven including the root) with $2^{5 \cdot n}$ potential nodes per level.

We choose x to be five so that we have seven levels, 0 - 6, where level 0 is the root node and level 6 contains all of the unique, executed instructions. Notice that nodes at any location in the tree are directly indexable by the program counter and require no comparison searching. Therefore all instructions can be accessed in constant time. The distribution table in Figure 2 shows what percentage of all allocated nodes are created at each level of the tree for the C programs. Approximately 95% of all of the nodes created are leaf nodes, which contain the static instructions. This indicates that the structure is composed primarily of useful nodes and is not causing an excess of intermediate nodes that do not store instructions.

An alternative approach is to create a hash table which is indexable by the program counter. This is ideal if every static instruction in an executable is executed and the table is the same size as the number of static instructions. However, we find that only 6% to 54% of static instructions are ever visited during the course of execution in several C, Fortran, and C++ programs [13]. Without prior knowledge of the number of unique executed static instructions, building a memory efficient hash table would not be possible. Even with this knowledge, it is inevitable that collisions will take place in the hash table, necessitating comparison searches to find the proper instruction. In fact, we find that about 10% of all accesses to a hash table equal to the size of unique static instructions results in a collision.



Distribution table of the tree and its nodes for C programs.

	Lvl 1	Lvl 2	Lvl 3	Lvl 4	Lvl 5	Lvl 6
% Nodes	0.0055	0.0063	0.0393	0.4281	4.9427	94.732
% Nodes Full	0.0	0.0	0.0	0.09	6.81	N/A
Avg	1.75	2.0	12.5	135	1570	31,771
Max	3 (vortex)	5 (vortex)	48 (vortex)	427 (vortex)	3964 (gcc)	68,585 (gcc)

Figure 2: Resurrection Tree and Distribution Table

2.2 Static and Dynamic Generation of Resurrected Code

There are several ways in which the resurrected code can be coupled with an existing trace-driven simulator. One choice is to create the resurrected code prior to running the simulation. Another choice is to create the resurrected code dynamically as the simulation is being performed.

Creating the resurrected code *before* doing the actual full-processor simulation requires two passes through the dynamic instruction stream. On the first pass through the stream, each instruction is interpreted and then placed into the resurrection tree on the first instance of the instruction. On subsequent instances of the instruction, the instruction need not be decoded.

Upon completion of the first pass, the resurrected code structure is complete and full simulation may take place (illustrated in Figure 1.b). The second pass through the dynamic stream progresses in a similar manner to traditional trace-driven simulation. Instructions no longer need to go through the entire decode process since all instructions decoded information can be accessed in the resurrected code using the program counter as an index.

The resurrected code may be stored in a file and used over in many simulations as long as the executable and data inputs do not change. This storage can be done in many different ways depending on the goals of the user. If an executable remains unchanged and will be run with several different inputs, the separately generated resurrected code information can be merged in an attempt to fill in some of the holes, providing better approximation of the object image.

In the case of trace based simulations, mispredicted branches can now be treated differently. On a mispredicted branch, the resurrected code structure becomes the new source of instructions until the branch target is resolved. When this happens, appropriate squashing takes place and the simulator resumes accepting instructions from the trace.

It is also possible to create the resurrected code *dynamically* as full-processor simulation is taking place. In this approach, instructions are placed into the resurrected tree and removed from the tree in the same manner as the beforehand approach. With the dynamic approach, there is no need to store resurrected code in a file format and reload it each time. There is a small accuracy tradeoff when implementing this approach. Branches that are mispredicted may be accessing a target that is not yet in the resurrected code, but will be at some time in the future. This type of mispredicted branch target is captured in the aforementioned static approach. This inaccuracy can be ameliorated by using an instruction buffer and dynamically performing the two-pass idea from the static approach.

3 Benefiting from Resurrected Code

There are two primary ways in which microprocessor simulation can benefit from the introduction of resurrected code. The first is decreased simulation time and the second is improved accuracy for trace based simulations.

3.1 Decreasing Simulation Time Using Resurrected Code

As ratios of dynamic instructions to static instructions clearly indicate, many unique static instructions are executed many times dynamically. Typically, a simulator decodes each instruction as it is encountered. So the processor may decode some static instructions many times. Much of the information being extracted is the same each time, such as opcode, logical registers, type of instruction, immediate values, instruction address, and other indicator bits. All of this information can be stored in the resurrection tree. When an instruction enters the decode stage, it can index the tree and check if this static information is already available due to an earlier decode of the same address. If so, the instruction does not need to be decoded further and can simply maintain a pointer to the static information in the tree.

It is not unusual that the instruction decode stage of simulation is the most time-consuming. This is the case in any style of simulator, including execution-driven and trace based simulators. Decode routines are typically comprised of unpredictable control transfer sequences (such as a C `case` statement) and may contain equally unpredictable pointers to lower-level decode functions. For these reasons, the impact of improving the performance of decoding can greatly affect simulation time as a whole.

For X86 processors like the Intel Pentium II, the decode cost per instruction is high. CISC instructions perform a multi-level decoding process where instructions are converted into one or more RISC-like micro-operations. These micro-operations are then executed by an out-of-order execution core. The decode operation can be accelerated by using the resurrected code data structure. Micro-operations can be stored in the resurrected tree and therefore eliminate the entire decode time for any future references to the same instruction. If this storage is too expensive, the first-level of decoding can be performed to determine the type of the CISC operation and a pointer to the second-level decode function can be stored in the resurrected code.

Like the micro-operation function pointer, other information associated with a particular static instruction can be stored in the tree, such as per-instruction statistics or hints to the simulator. Dynamic information, like effective addresses, branch decisions, and indirect targets, is unique to each dynamic instruction and will need to still be acquired on each reference to an instruction. Storing them in the resurrected tree is an efficient way to handle it. Finally, if there is self-modifying code in a program, an additional check must be performed to guarantee that the opcode has not changed.

3.2 Improving Accuracy of Mispredicted Path Simulations

In trace-driven timing simulation, the trace provides information on whether or not a branch is taken in the actual flow of execution. If the prediction from the branch predictor contradicts the actual resolution of the branch, then the resurrected code can be accessed. In many cases, the code found in the resurrected tree can pass through the simulator like any other instruction. The exceptions are memory access instructions, like loads and stores, and indirect control transfers.

Memory Accesses A unique load instruction in an executable may be executed many times. Each time, it is possible for the load to have a different address based on the source registers and flow of control. This address has to be provided by the trace since it cannot be calculated by a non-functional simulator. When a load is encountered in the resurrected code, the trace is no longer providing correct information about the instruction and the effective address is unknown. Some of the most important effects of executed wrong path instructions are the effects on the data cache. So it is important to make a good guess at the correct address for a misspeculated load or store instruction.

Many of the current value and address prediction techniques [14, 15, 16] could be used to help effectively "guess" or predict the wrong path memory access addresses. For example, if some load or store instruction previously accessed the following four addresses: `2000, 2004, 2008, 2012`, then we can predict with high confidence that the next address will be `2016`. Another high confidence situation results when the previous addresses are all identical or the effective address is static. In this case, we can predict that the address will remain the same. For architectures that make heavy use of a stack, such as the X86 architecture, there are stack based patterns that can be exploited as well. For addresses that do not fall into these three categories, a slightly more random guess can be made based on a previous history of addresses. Some examples are randomly picking one of the previous addresses, choosing the most frequently accessed address, or deciding based on a weighted average. Any of these techniques are acceptable due to the nature of non-functional, trace based simulation. Correct data values are not necessary to proceed, so the focus is on accessing the correct *areas* of the memory.

Indirect Control Transfers A similar problem as the memory access problem arises on indirect branches, jumps, and calls. The target for these instructions is provided by the trace along the correct path, but if an indirect control transfer is encountered along the mispredicted path then the simulator does not have enough information to proceed. Once again, a prediction needs to be made for the target. Luckily, in most programs, indirect control transfers do not have many targets and it quickly becomes apparent which target to choose. When this is not the case, like in object-oriented programs which contain many virtual functions, the control transfer can be selected in a more random manner similar to the memory access guesses. Branch target prediction mechanisms employed by processors can also be used.

Incomplete Sequences from Resurrected Code The resurrected code is not a complete copy of the original object code segment. Some portions of the executable are never executed, such as error code and code not reached due to the input set. It is possible for the availability of resurrected code instructions along a wrong path to end up (or start) at a "hole" in the resurrected code. Upon reaching a hole, there are several choices for the simulator. The best choice is not as obvious as with the case of memory references and control transfer targets.

The instruction immediately before a hole is often a conditional branch. When this is the case and the simulation reaches a hole in the resurrected code along the wrong path, it can go back and take the alternate path for the preceding branch. At this point it will find some code and can continue executing until the branch that was misspeculated is resolved. This is not the path that the processor predicts, but it is important to continue to execute instructions while the mispredicted branch is being resolved. Another considered approach that can be used with or instead of the previous approach involves marking the first instruction in the current basic block. When a wrong path instruction is not found in the resurrected code, then simply return to the head of the basic block. This forms a loop until the branch is resolved. The last set of options consists of stalling the processor or claiming the branch resolved and returning to the normal execution stream.

4 Simulation Methodology

We perform trace-driven simulation to determine the impact of the proposed resurrected code structure. Our simulation environment consists of a trace generated on a Sun UltraSPARC, a cycle-level, full microprocessor simulator, and several benchmark suites.

4.1 Tracing and Profiling Tools

Traces are generated dynamically using the tool Shade [8] on a Sun UltraSPARC-II processor. Shade is a tool that dynamically executes and traces SPARC executables. The traces represent the retired instruction stream and do not contain register values. Shade is customizable and allows the user to specify the exact trace information to collect. At these points, the trace information can be dynamically handled in any manner. Shade only traces user and library code and does not analyze kernel code. However, if we were to use traces that include operating system effects, our resurrected code would handle it properly.

Detailed information can be collected dynamically for every instruction and opcode. We collect data such as the opcode fields (to identify type of branches), program counter, branch targets, and taken/not-taken branch information. This information is then processed by our own software simulation tools.

4.2 Benchmarks

We use programs from SPEC CINT95, SPEC CFP95, SPEC JVM98, as well as a suite of C++ programs. Short descriptions of all the benchmark programs are in Table 1. Our C++ suite has been used to study the behavior of C++, specifically the cost of virtual functions calls [17] [18]. Three programs from the floating-point SPEC, CFP95, are analyzed for our study. These programs are computation-intensive and written in Fortran. The SPEC JVM98 benchmark suite [19] is a series of Java programs supplied as Java byte code. These programs are run using the Java Virtual Machine (JVM) version 1.1. Multiple threads are simulated within the JVM and not run natively on the operating system. The traces studied for JVM98 are the dynamic instructions produced by the JVM interpreting and executing the Java byte code of the benchmark programs.

Table 1: Benchmark Descriptions

Program	Input/Flags	Description of Program
SPEC CINT95: C programs		
compress95	test.in	Compresses large text files
gcc	amptjp.i	Compiles pre-processed source
go	2stone9.in	Plays the game Go against itself
li	train.lsp	Lisp interpreter
m88ksim	-c ctl.in	Simulates the Motorola 88100 processor
perl	scrabbl.pl scrabbl.in	Performs text and numeric manipulations
SPEC JVM98: Java Programs		
compress	-s1	A popular LZW compression program.
jess	-s1	NASA's CLIPS rule-based expert systems.
db	-s1	Data management benchmarking software from IBM.
javac	-s1	The JDK Java compiler from Sun Microsystems.
mpegaudio	-s1	The core algorithm decoding an MPEG-3 audio stream.
jack	-s1	A real parser-generator from Sun Microsystems.
Suite of C++ Programs		
deltablue	3000	Incremental dataflow constraint solver
eqn	eqn.input.all	Type-setting program for math. equations
idl	all.idl	SunSofts IDL compiler 1.3
ixx	object.h Som_Plus_Fresco.idl	IDL parser generating C++ stubs
richards	1	Operating system simulation benchmark
SPEC CFP95: Fortran Programs		
fpppp	natoms.in	Performs multi-electron derivatives
hydro2d	hydro2d.in	Hydrodynamical Navier Stokes equations
tomcatv	tomcatv.in	Generation of 2-D coordinate system

4.3 Full Simulation

To analyze the impact of the resurrected code in a full processor environment, we use a detailed, trace-driven, cycle-level, timing simulator that models all resource contention as well as speculative execution. Shade is the front-end of the simulator. It takes any SPARC executable (source code not necessary) as input and then drives the execution core with a dynamic stream of instructions. Therefore, the simulator uses the SPARC instruction set architecture [20] and handles the SPARC nuances in a proper fashion (e.g. register windows, conditional instructions, condition code registers, delay slots). While Shade executes the program with functional correctness, the simulation core does not simulate register-level passing of data and is therefore only a cycle-by-cycle timing simulator.

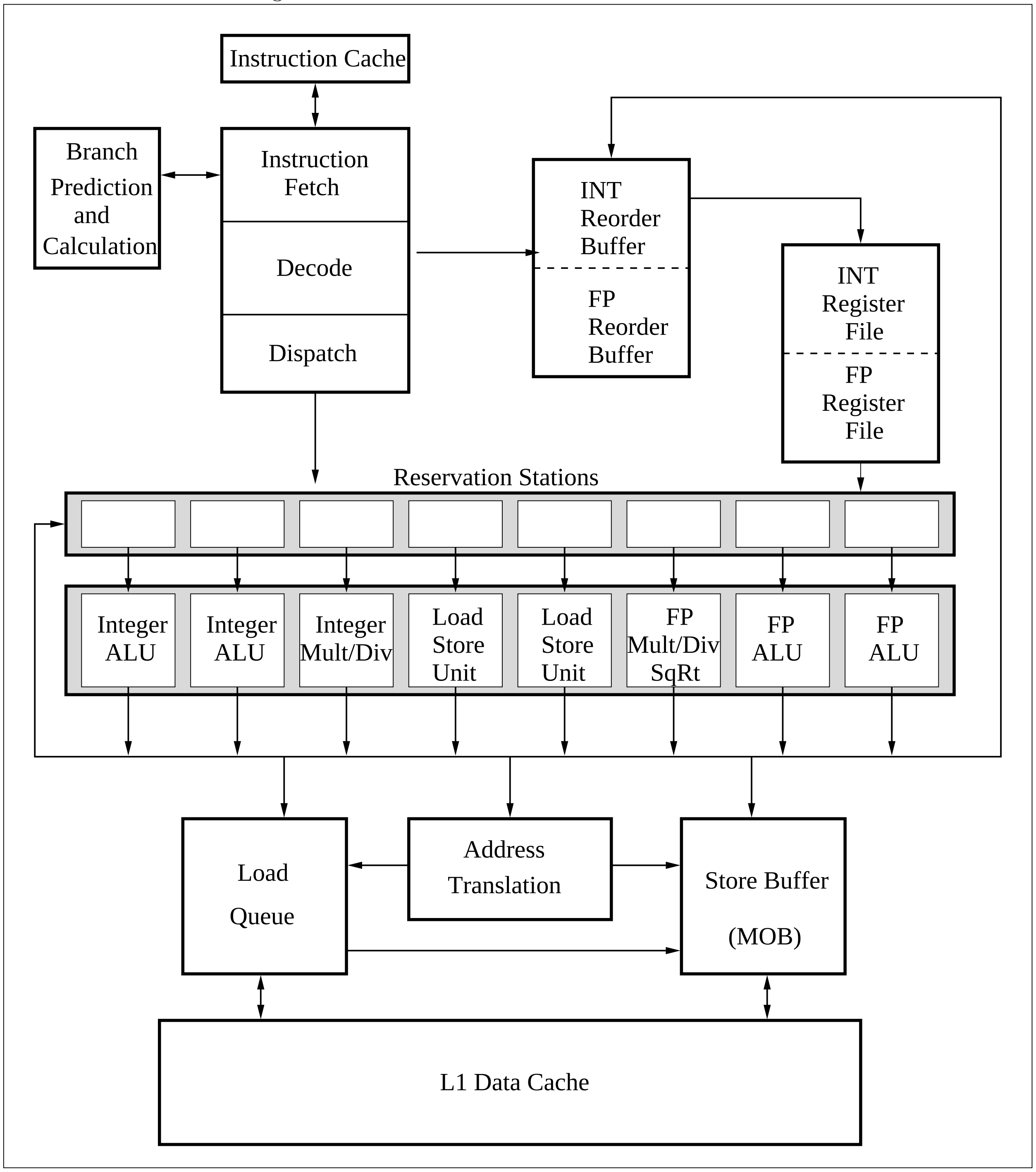
For this study, the base architecture model is loosely based on a combination of the Sun UltraSPARC-II microarchitecture and features from the SimpleScalar *sim-outorder* default simulation model [2]. The model is a four-wide machine, i.e. four-wide issue, decode, and retire. This model is shown in Figure 3.

The execution core of the base model contains two basic integer ALU's with one cycle latency and one cycle throughput (represented by 1-1), one integer multiply/divide unit (3-1 for multiply, 20-19 for divide), two basic floating point (FP) ALU's (2-1), one FP multiply-divide-square root unit (3-1 for multiplies, 12-12 for divides, 24-24 for square root), and two load-store units (1-1). Each of these units is supplied by an eight entry, first-available reservation station. The load-store unit calculates effective addresses and then dispatches the loads and stores to the proper location - load queue or store buffer. Branch addresses are calculated in the decode stage if possible. The simulator uses a separate 48-entry reorder buffer (ROB) and register file for floating point and integer instructions, as in the UltraSPARC. Stores are allocated entries in the ROB.

Our study requires the simulation of dynamic branch prediction hardware in order to identify when misspeculation occurs. We use the Gshare branch prediction scheme as described by McFarling in [21]. The primary predictor is 2048 entries, direct-mapped, and indexed by the program counter plus five global history bits. This predictor is accompanied by a 512-entry, direct-mapped branch target buffer (BTB) to predict target addresses for the predicted branches.

We are most interested in branches that begin fetching from the wrong target address due to misprediction. There are two cases in which this happens and a branch misprediction is reported. One case surfaces when Gshare mispredicts the branch. The other occurrence is when the Gshare method correctly predicts a branch is taken, but the branch instruction's target address is not correct in the BTB. If Gshare predicts the branch is not taken, then the BTB is not consulted.

Figure 3: Overview of Simulated Microarchitecture



The cache hierarchy simulation model is derived from `cachesim5` which is available with the Shade tool set. The L1 instruction cache is a 16 KB, two-way set-associative, write-through cache with a block size of 32 bytes, a hit latency of one cycle and uses the LRU replacement algorithm. The L1 data cache is a 16 KB, four-way set-associative. write-through cache with a block size of 32 bytes, a hit latency of one cycle and uses the LRU replacement algorithm. The L2 cache is a unified, 1 MB, four-way set-associative, write-back, write-allocate cache with a block size of 64 bytes, a hit latency of six cycles, and uses a random replacement scheme. Address translations are given a constant latency of one cycle (assuming TLB hits or infinite main memory).

Finally, the resurrected code is implemented as well. The resurrected code is created beforehand to prevent the slight disadvantage that results from dynamic creation. When the simulator is executing along the mispredicted path and encounters an instruction that is not in the resurrected code, no more instructions are fetched from the resurrected code. This allows us to evaluate the effectiveness of the resurrected code as a source for wrong path instructions.

5 Impact and Analysis

Table 2 reports some characteristic of the benchmarks that are studied. *Total Instr* are the total number of instructions issued by the simulator. This includes wrong path and retired instructions. Long running simulations are ended at approximately 250 million instructions. *% RC Instr* are the percentage of the *Total Instr* that are wrong path instructions from the resurrected code. *% Branches* are the percentage of the *Total Instr* that are branches. *% Mispredict* represents the percentage of all the branches that are mispredicted by the branch prediction unit of the simulator. The misprediction rate along with the percentage of branches relates directly to the number of times that resurrected code needs to be accessed.

The impact of the resurrected code on the execution of a program is best shown with the resurrected code instruction percentage in Table 2. When this percentage is low, traditional timing simulators are not suffering significant accuracy losses, but once this percentage begins to grow, approximating the impact of wrong path instructions is no longer advisable. This table shows that when using the resurrected code structure, the percentage of wrong path instructions ranges from less than one percent in the Fortran code to 27% in the C++ code. This percentage is not directly related to any one characteristic of the program. It is a function of many characteristics - the number of branches, the misprediction rate, the average time to resolve a branch, and the number of wrong path instructions available in the resurrected code. Remember that for this study of resurrected code effectiveness, we are terminating the wrong path access if once an instruction along the mispredicted path is not available in the resurrected code.

Table 2: Benchmark Characteristics

Benchmark	Total Instr	%RC Instr	%Branches	%Mispredict
gcc	255M	3.89745	17.9024	11.1949
li	172M	3.12769	17.2188	7.17111
compress95	42M	8.66767	10.9306	8.61377
m88ksim	124M	1.71845	13.6253	2.6591
go	273M	10.3733	13.3304	22.4224
perl	42M	3.08663	15.8848	10.2973
compress	250M	1.34541	8.21657	4.52915
db	88M	3.21797	14.0275	7.49158
jack	251M	1.61124	10.1666	5.24806
javac	204M	3.00627	13.525	6.90501
jess	253M	3.07519	12.8812	7.07266
mpegaudio	251M	1.23916	7.75227	4.6606
deltablue	41M	25.4161	6.22501	12.3228
eqn	48M	17.3604	9.42446	17.4756
idl	85M	21.8395	2.65526	16.4061
ixx	30M	15.1617	7.74375	15.1446
richards	67M	27.4873	8.17629	18.1409
fpppp	243M	0.101341	1.29608	8.69944
hydro2d	247M	0.533453	14.268	2.75581
tomcatv	247M	1.23992	17.2669	5.16266

Total Instr are the total number of instructions issued, including wrong path instructions. *% RC Instr* are the percentage of the *Total Instr* that are from the resurrected code. *% Branches* are the percentage of the *Total Instr* that are branches. *% Mispredict* represents the percentage of all the branches that are mispredicted.

Table 3 illustrates a more in-depth look at how the resurrected code is performing. The table presents the *average run length* which is the number of wrong path instructions executed from the resurrected code before a branch is resolved or an instruction is not found in the resurrected code. The next column of the table shows what percentage of wrong path accesses into the resurrected code are completely successful, *% Complete Runs*, and the fourth column shows what percentage are cut short by a missing instruction in the resurrected code, *% Incomplete Runs.* These numbers are also shown graphically in Figure 4. Along with these percentages in the table are the average run lengths of the successful accesses (column 3) and the incomplete accesses (column 5).

Table 3: Characteristics of Wrong Path Accesses

Benchmarks	Avg Run Length	% Complete Runs	Complete Avg	% Incomplete Runs	Incomplete Avg
gcc	1.53282	96.2352	1.49234	3.76483	2.56746
li	2.25462	98.6498	2.25251	1.35023	2.40866
compress95	5.75522	97.4103	5.67324	2.58968	8.83877
m88ksim	2.88976	92.1436	2.62136	7.8564	6.03765
go	3.16642	95.6717	3.22277	4.32826	1.92079
perl	1.0593	93.4304	0.825984	6.56956	4.37745
compress	2.15051	95.6985	2.00174	4.30148	5.4604
db	2.37987	94.3183	2.2362	5.68169	4.76482
jack	2.30996	95.3751	2.20628	4.62494	4.44801
javac	2.57011	96.7709	2.49351	3.22913	4.86571
jess	2.66014	96.0798	2.5856	3.92018	4.48706
mpegaudio	2.43668	97.2755	2.35058	2.7245	5.51094
deltablue	4.59549	99.1978	4.60399	0.802208	3.54477
eqn	1.49341	96.4448	1.37921	3.55524	4.59146
idl	2.99445	96.9867	2.86113	3.01328	7.28563
ixx	1.60074	96.5479	1.42273	3.45208	6.57946
richards	0.758869	99.9954	0.758727	0.0046	3.87195
fpppp	0.848098	98.6521	0.846684	1.34786	0.951604
hydro2d	1.35015	96.6171	1.39515	3.3829	0.0648255
tomcatv	1.3725	98.2111	1.38713	1.78892	0.569438

Notice that in all cases, over 92 percent of resurrected code accesses are completed successfully. This indicates that the resurrected code does a competent job of approximating the object code image. It is interesting to notice that the percentage of complete accesses is not directly related to any one of the characteristics presented in this study. Successful accesses are not a function of average run length, misprediction rate, number of branches, or of the percentage of static code instructions that are touched [13]. Instead, there is some path-based redundancy property associated with a program where programs have a tendency to execute both paths of a working set of branches and then frequently execute along those paths.

Another observation is that wrong path accesses that do not complete successfully are longer than accesses that do complete successfully. The longer the simulator remains in the resurrected code, the chance of encountering a hole increases. It is interesting to look at the distribution of the lengths of the wrong path resurrected code accesses. If the number of sequential wrong path accesses brought into a machine shows a low deviation from the average, then uniform penalties may well be a good way to approximate the wrong path effects.

Figure 4: Percentage of Complete and Incomplete Wrong Path Resurrected Code Accesses

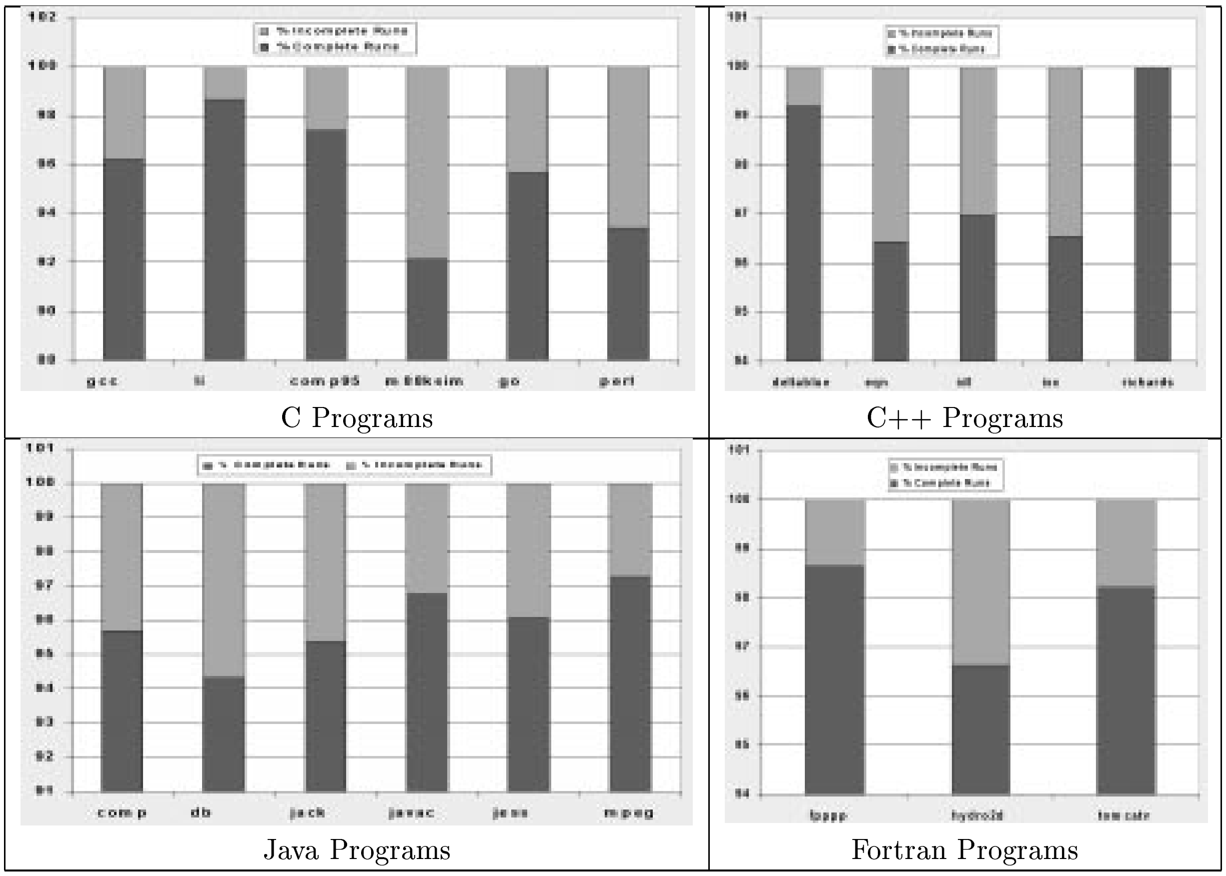
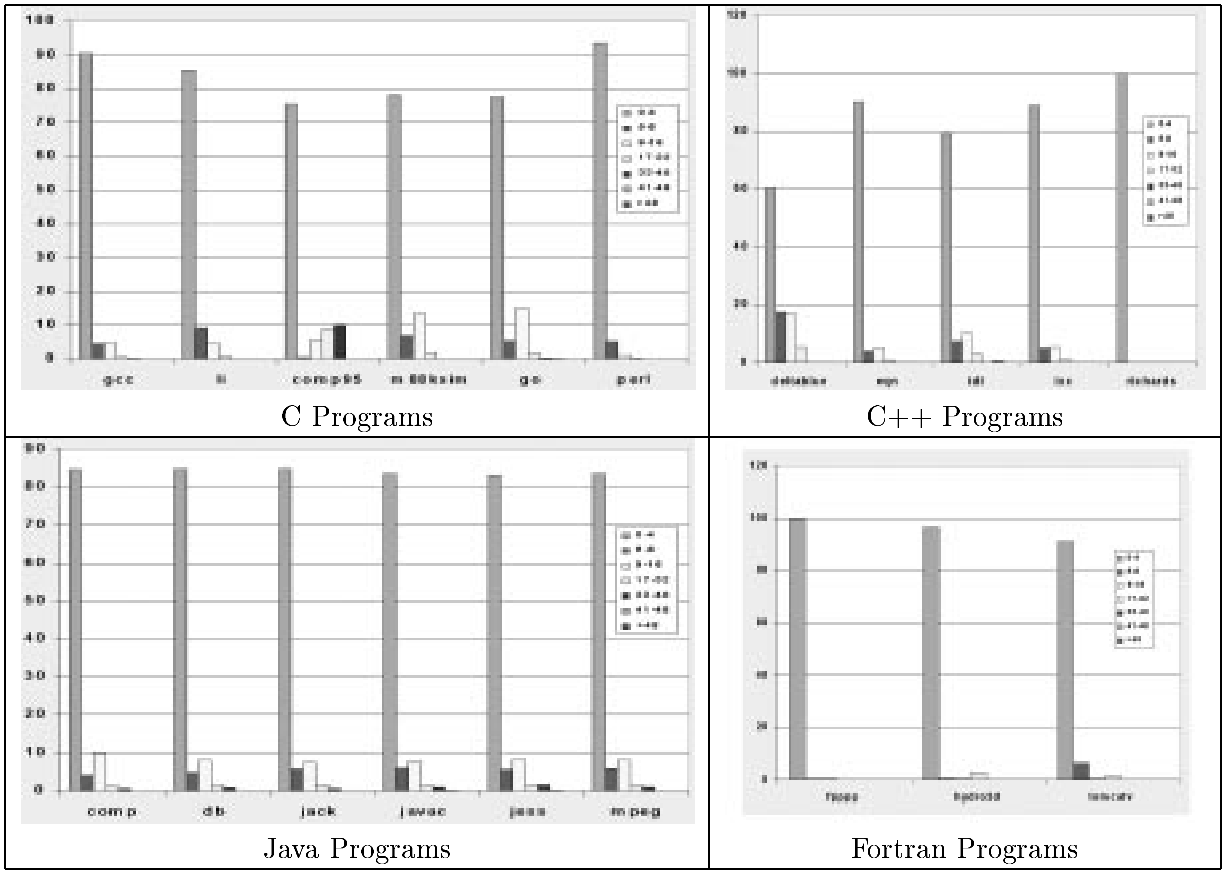


Figure 5 shows a breakdown for each benchmark of the number instructions found in the resurrected code following each mispredicted branch. Note that the access lengths are fairly regular for the Java and Fortran program and a majority of the accesses are short, in the less than four instruction range. For the C and C++ programs, this is not the case. The number of instructions that are introduced into the simulation due to the use of resurrected code along the wrong path varies greatly. This indicates that a uniform misprediction penalty is not going to model these programs properly. These wrong path accesses to the resurrected code can be ended due to a mispredicted branch becoming resolved or a hole in the resurrected code. The access lengths are therefore a property of the resurrected code, program behavior, and processor model.

Figure 5: Wrong Path Access Length for All Accesses



C Programs

C++ Programs

Java Programs

Fortran Programs

Table 4 reports the instruction mix of the wrong path instructions fetched from the resurrected code. The classes of instructions represented in the table are instructions that have the most impact on the processor. All instructions consume resources such as physical registers and reservation station entries. Loads and stores cause clutter of the load queue and store buffer. Loads may even proceed far enough that they access the cache. In these benchmarks, as much as 5% of wrong path loads make a cache access.

Table 4: Instruction Mix of Wrong Path Instructions Fetched from Resurrected Code

Benchmark	% Loads	% Stores	% Branches	% Loads to Cache
gcc	23.2795	04.6946	12.5186	4.267
li	31.9055	12.9008	15.2739	2.868
compress95	16.5678	10.9529	13.813	4.444
m88ksim	12.28	03.00082	18.661	3.619
go	15.8291	07.47483	12.5616	5.492
perl	19.3525	06.88849	13.4495	1.712
compress	18.6642	02.39492	15.4584	3.662
db	11.9013	03.52667	20.1537	4.022
jack	13.3168	04.38963	18.3014	4.429
javac	10.6024	03.61854	21.095	3.912
jess	13.7517	03.54733	19.6948	4.540
mpegaudio	12.9837	03.23379	20.055	4.048
deltablue	11.7917	03.94386	12.8409	2.321
eqn	12.674	03.47143	18.735	3.147
idl	14.848	06.51755	17.4596	4.796
ixx	19.9524	05.2864	12.2186	3.450
richards	33.0823	07.1196	00.119921	0.0
fpppp	40.1704	24.4148	02.22202	1.358
hydro2d	20.3805	04.75228	12.616	2.473
tomcatv	29.0301	07.56081	09.82348	1.814

6 Conclusion

We introduce a structure, called *resurrected code*, to improve the accuracy of mispredicted path simulations and decrease the simulation time of microprocessor software simulators by creating an approximates copy of the object code. The resurrected code can be created before running full processor simulation or during the simulation. By storing static decoding information and simulator-specific hints in the resurrected code, the simulator does not need to replicate costly decoding and instruction initialization work.

The resurrected code can be easily used to increase the accuracy of trace based simulation. Non-functional, timing simulation allows for a wider selection of programs to simulate, but traditionally does not model speculation with high degrees of accuracy. The resurrected can be used as a source for speculative fetching and execution along mispredicted paths. The handling of memory access instructions, and control transfer instructions is addressed. We run simulations on a full-processor, cycle-level trace-driven simulator and analyze the impact of allowing a timing simulator to introduce instructions along mispredicted paths. We find that up to 27% of instructions in a program can be wrong path instructions that are fetched from the resurrected code. We also find that only 1% to 7% of accesses to the resurrected code do not complete due to lack of instructions. These results indicate that in addition to reducing the simulation time in a variety of simulation frameworks, the resurrected code can significantly improve the realistic modeling of speculative microprocessors within a simple, trace based simulation framework.

References

[1] P. Bose and T. M. Conte, "Performance Analysis and Its Impact on Design," *IEEE Computer*, pp. 19–22, May 1998.

[2] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Tech. Rep. CS-TR-96-1308, University of Wisconsin, Madison, WI, July 1996.

[3] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM Reference Manual. Version 1.0.," Tech. Rep. 9705, Department of Electrical and Computer Engineering, Rice University, July 1997.

[4] B. Grayson, "Armadillo: A High-Performance Processor Simulator," Tech. Rep. TR-PDS-1996-008, The University of Texas at Austin, May 1996.

[5] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Santa Fe, New Mexico), pp. 4–11, May 1988.

[6] T. M. Conte and C. E. Gimarc, *Fast Simulation of Computer Architectures.* Kluwer Academic Publishers, 1995.

[7] SpeedTracer, a hardware trace generator system for the X86, AMD, Austin.

[8] R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," Tech. Rep. SMLI 93-12 and UWCSE 93-06-06, Sun Microsystems Laboratories, Incorporated, and the University of Washington, 1993.

[9] M. Reilly and J. Edmondson, "Performance Simulation of an Alpha Microprocessor," *IEEE Computer*, pp. 50–58, May 1997.

[10] A. Paithankar, "AINT: A Tool for Simulation of Shared-Memory Multiprocessors," Master's thesis, University of Colorado, Boulder, Colo., 1996.

[11] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models," *IEEE Computer*, vol. 31, pp. 59–65, May 1998.

[12] M. Moudgill, J. Wellman, and J. E. Moreno, "An approach to quantifying the impact of not simulating mispredicted branches," *Workshop Digest of the PAID Workshop held in conjuction with ISCA98*, pp. 60–66, July 1998.

[13] R. Bhargava, L. K. John, and F. Mathus, "Accurately Modeling Speculative Instruction Fetching in Trace-driven Simulation," in *Proc of Int. Performance, Computing, and Communications Conference*, Feb 1999. To appear.
http://www.ece.utexas.edu/ ravib/papers/tracefetch.pdf.

[14] G. S. Tyson and T. M. Austin, "Improving the Accuracy and Performance of Memory Communications Through Renaming," *Proc. 30th Intl. Sym. on Microarchitecture*, pp. 218–227, Dec. 1997.

[15] A. Moshovos, S. Breach, T. N. Vijaykumar, and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," in *Proc. 24th International Symposium on Computer Architecture*, pp. 181–193, June 1997.

[16] Y. Sazeidis, S. Vassiliadis, and J. Smith, "The Performance Potential of Data Dependence Speculation and Collapsing," in *Proc. 29th International Symposium on Microarchitecture*, pp. 238–247, November 1996.

[17] K. Driesen and U. Holzle, "The Direct Cost of Virtual Function Calls in C++," in *OOPSLA-96*, (San Jose, Calif.), pp. 306–323, Oct 1996.

[18] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs," Tech. Rep. CU-CS-698-94, University of Colorado, Boulder, Jan 1994.

[19] Standard Performance Evaluation Corporation, "SPEC JVM98 Benchmark."
http://www.spec.org/osg/jvm98/.

[20] D. L. Weaver and T. Germond, *The SPARC Architecture Manual (Version 9).* Sparc International, Englewood Cliffs, NJ, USA, 1995.

[21] S. McFarling, "Combining Branch Predictors," Tech. Rep. TN-36, Digital Western Research Labs, Palo Alto, Calif., Jun 1993.