*Appears in the International Conference on Parallel Architectures and Compilation Techniques (PACT)*
*Brasov, Romania, September, 2007*

# Error Detection Using Dynamic Dataflow Verification

**Albert Meixner**
*Dept. of Computer Science*
*Duke University*
*albert@cs.duke.edu*

**Daniel J. Sorin**
*Dept. of Electrical and Computer Engineering*
*Duke University*
*sorin@ee.duke.edu*

## Abstract

*A significant fraction of the circuitry in a modern processor is dedicated to converting the linear instruction stream into a representation that allows the execution of instructions in data dependence order, rather than program order, to extract instruction level parallelism. All errors caused by hardware faults in this circuitry—which includes the fetch and decode stages, renaming and scheduling logic, as well as the commit stage—will manifest themselves as incorrectly constructed dataflow graphs.*

*Dynamic Dataflow Verification (DDFV) compares the dynamically constructed and executed dataflow graph to the expected dataflow graph of the static program binary, represented by a signature embedded in the instruction stream. The signature comparison enables comprehensive detection of transient errors, permanent errors, and design bugs in the dataflow circuitry. We show that DDFV detects errors with high probability, at a low hardware and performance cost.*

## 1. Introduction

As CMOS technology continues to scale, it becomes more susceptible to errors due to transient and permanent hardware faults [29, 12]. In this work we contribute an inexpensive mechanism for detecting errors in parts of the processor that previously required hardware replication or temporal redundancy for error detection. Our error detection scheme can be used for low-cost protection of a significant fraction of the core area or to reduce the area and performance costs of other previously proposed schemes. We focus on error detection and do not investigate the orthogonal problem of error recovery, which has been researched extensively in prior work (e.g., [9]).

The mechanism presented in this paper, *dynamic dataflow verification (DDFV)*, detects hardware errors by verifying at runtime that the dataflow graph specified by the program is the same as the dataflow graph being reconstructed and executed by the processor.

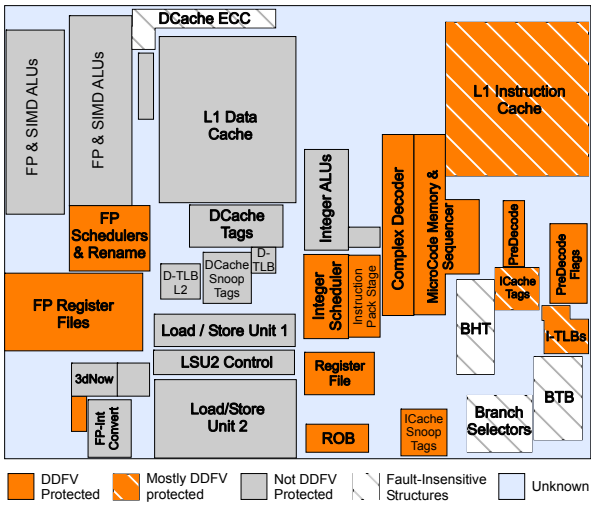



**Figure 1. DDFV coverage area in an AMD K8**

Because the process of dynamically reconstructing the dataflow graph involves so many of the superscalar processor's components—including the logic for fetch, decode, register rename, register read, writeback, and commit—DDFV can detect errors in a large fraction of the units within the core (illustrated in Figure 1). Furthemore, DDFV will inherently detect faults in the instruction cache logic and I-TLB, including errors in cache decoders and tag comparators that are not covered by error detecting codes, because errors in fetched instructions also alter the dynamic dataflow graph.

DDFV provides error detection by dynamically verifying a high-level invariant that an error-free system is guaranteed to maintain, rather than by adding low-level error detection in individual circuits. This high-level approach avoids coverage holes and leaves the design of lower-level components unchanged.

The following section (Section 2) discusses prior work in error detection and how it compares to DDFV. Section 3 specifies the system model and error model that we assume throughout the high-level overview of DDFV (Section 4), the discussion of its implementation details (Section 5), and the evaluation of DDFV's error coverage and performance impact (Section 6).

Finally, we describe how our proposed technique can be combined with other techniques to provide low-cost error detection for different systems (Section 7), and we draw conclusions about this work (Section 8).

## 2. Related Work

DDFV detects errors due to transient and permanent faults, as well as many design bugs, that cause the dataflow in the executed program to diverge from the correct dataflow specified in the binary. Prior schemes that cover the same space as DDFV use error detecting codes (EDC), temporal redundancy (re-execution), structural redundancy, built-in self-test (BIST), or control flow checking.

**EDC.** EDC is efficient for detecting errors in data values, particularly in storage and communication, and it is complementary to DDFV. In fact, our implementation of DDFV uses EDC to detect errors in the *values* in the dataflow graph (in the register file, ROB, and bypass network). The rest of our DDFV implementation detects errors in the *shape* of the dataflow graph.

EDC, by itself, has often been considered sufficient for non-safety-critical processors, because storage has been more susceptible to errors than logic. Researchers project, however, that logic errors will become more prominent [27].

**Temporal Redundancy.** A popular variant of temporal redundancy is redundant multithreading (RMT) [25, 23, 19], which detects transient errors by comparing the results of redundant threads. Permanent faults can be detected only in units that are replicated such that the two threads can use different copies. This is not typically the case for the front-end components and rename logic covered by DDFV. A detailed analysis of an RMT scheme showed that its performance degradation was about 30% [19]. RMT also incurs a significant increase in energy consumption, but the hardware costs of enabling RMT in a multithreaded processor are low. Another approach to temporal redundancy is software-based replication of instructions [22, 24]. These schemes detect transient errors but cannot detect errors due to permanent faults in many components. Software is cheap and flexible, but comes at the cost of a 50% slowdown [24] and high energy consumption.

**Structural redundancy.** Dual modular redundancy (DMR) and other structural redundancy schemes detect virtually 100% of all possible errors by running all operations on two copies of a component and comparing the results. Replication can be performed at different granularities (units vs. cores), but always comes at a considerable hardware cost.

DIVA is a heterogeneous DMR scheme [5,32] that uses a simplified, yet functionally identical, core for checking. This heterogeneous design reduces the hardware cost, as compared to homogeneous DMR, without sacrificing error coverage. Heterogeneous DMR is well-suited for large speculative RISC machines, such as the Alpha 21264, where it incurs virtually no slowdown and only 6% area overhead for the checker core [32]. However, it is less efficient for processors that utilize little speculation and simple dependency tracking—e.g., VLIW (Itanium [18]), SMT (Ultrasparc T1 [15]), DSPs (TI CMS320C54x [10])—because the complexity gap between the primary and checker cores is much smaller. DIVA is also less suitable for units that are inherently complicated and hard to simplify, such as fetch and decode on CISC machines, especially modern ones that crack instructions into micro-ops (like the AMD K8 [1] and Intel Pentium4 [8]). DDFV does not have these problems, because its cost is mostly independent of the complexity of the verified hardware and lower than even a simple replicated unit.

**BIST.** BIST has long been used to detect defects during start-up. Recently BIST has also been used in the BulletProof pipeline [28] for runtime detection of permanent faults. BulletProof detects and diagnoses 89% of the possible permanent faults in a VLIW processor model at the cost of 5.8% extra hardware and with negligible performance impact. However, it offers no protection from transient faults.

**Control flow checking.** Control flow checkers [7, 14, 31] dynamically verify that a program is following a legal path of execution. Some checkers [31] only verify inter-block control-flow, and are thus complementary to DDFV, which implicitly checks intra-block control-flow. Other control flow checkers also check decode signals generated during execution [7,14] and are closer in nature to DDFV as they also capture faulty instruction decoding and thus provide overlapping functionality. However, all of the latter schemes target strictly in-order processors and do not address errors in propagating instruction results or instruction scheduling. Control flow checkers have negligible hardware cost and low performance impact.

**Summary.** Most of these prior mechanisms at least partially overlap with DDFV and most exhibit weaknesses that can be mitigated by combining them with DDFV (see Table 1 for a summary). We discuss some attractive possible combinations in Section 7.

## 3. System Model and Fault Model

DDFV verifies that the execution matches an abstract model, rather than checking the correct opera-

**Table 1. Error Detection Coverage Comparison.** Entries in the table represent detection for T(ransient), P(ermanent), and D(esign) errors. Shaded entries correspond to no *significant* error detection coverage

| | Component/Activity | DDFV | control flow checking [7, 14] | DMR, DIVA [5,32] | redundant multi-threading [19] | software redundancy [24] | BIST, Bullet-Proof [28] |
|---|---|---|---|---|---|---|---|
| *Frontend* | fetch logic | T, P, D | T, P, D | T, P, D | T | T | P |
| | decode logic | T, P, D | (T, P, D)[a] | T, P, D | T | T | P |
| | reg. rename/read logic | T, P, D | | T, P, D | T, P | T, P | P |
| *Scheduling* | reorder buffer | T, P, D | | T, P, D | T, P | T, P | P |
| | reservation stations | T, P, D | | T, P, D | T, P | T, P | P |
| | load-store queue | | | T, P, D | T, P | T, P | P |
| *Execution* | ALUs | | | T, P, D | T, P[b] | T, P[b] | P |
| | FPUs | | | (T, P, D)[c] | T, P[b] | T, P[b] | P |
| | multiplier and divider | | | T, P, D | T, P[b] | T, P[b] | P |
| | branch units | (T, P, D)[d] | (T, P, D)[e] | T, P, D | T, P[b] | T, P[b] | P |
| | load-store unit | | | T, P, D | T, P[b] | T, P[b] | P |
| *Backend* | PC update logic | (T, P, D)[d] | (T, P, D)[e] | T, P, D | T, P[b] | T, P[b] | P |
| | register file write logic | T, P, D | | T, P, D | T, P | T, P | P |
| *Cost* | area | low | very low | low-high | very low | none | low |
| | performance | low | low | low | medium-high | high | none-low |
| | power | low | very low | low-high | medium | medium | low |

a. Only if control flow checksum is computed over control signals emitted by instruction decoder (e.g. [14]). All such proposed schemes have only considered in-order cores.

b. Can detect permanent faults if original and redundant instruction use different instances of this resource.

c. Not detected by proposed DIVA implementations, but could be detected with straightforward extensions.

d. Only if incorrect branch target or corrupted PC is not at beginning of basic block. Does not guarantee legal path.

e. Only if incorrect branch target or corrupted PC is not on a legal path. Cannot detect incorrect branch decision.

tion of any specific component, and it is therefore not tied to a specific architecture or error model. Nevertheless the details of our DDFV implementation and its error detection capabilities depend on the targeted system and error model.

**System Model.** We target superscalar, dynamically scheduled processors, such as the Intel Pentium4 [8], AMD K8 [1], etc. The focus on superscalar processors is not rooted in a fundamental limitation of DDFV, which can be applied to other types of processors, but reflects the dominant architecture. DDFV is strictly limited to dataflow within a single thread of execution and is therefore oblivious to the presence of multiple cores or thread contexts.

**Error Model.** We consider errors due to transient and permanent hardware faults (stuck-at-0 and stuck-at-1) in all structures in the microprocessor core. By comparing hardware operation to an abstract model of correct behavior, dynamic verification schemes can also detect many design bugs[1], which cause hardware to behave incorrectly despite the absence of physical errors. Schemes that use identical hardware for execution and verification (such as RMT and standard DMR) and schemes that verify hardware implementations rather than abstract behavior (such as BIST and Bullet-Proof) cannot detect design bugs. Recent microprocessors from Intel and AMD have been shipped with dozens of design bugs [2, 11], and online mechanisms to uncover them can be a valuable tool in processor development and verification.

## 4. High-Level Overview of DDFV

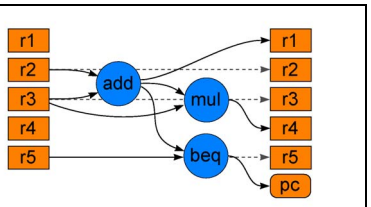
The basic idea behind DDFV is to periodically compare the static dataflow graph in the program binary to the actual dataflow in the processor during execution. To implement this idea we need three things: an efficient representation of a dataflow graph that can be easily compared, a way to statically compute this rep-

1. We cannot quantify the coverage of design bugs, because there is an unbounded number of possible design bugs.

resentation and attach it to the program, and a mechanism to track dataflow between the instructions dynamically executed in the processor.

### 4.1. Dataflow Graph Representation

The full dataflow graph (DFG) of a program is not known at compile time, because it depends on dynamically computed branches. To still be able to compare the entire program execution to a dataflow graph statically embedded in the program binary, DDFV partitions the program into blocks of code that have statically known dataflow graphs, such as the simple example in Figure 2. After each block completes, DDFV verifies that the dataflow graph was executed correctly and moves on to verify the next block.




```
add r1, r2, r3
mult r4, r1, r3
beq r5, r1, PC+9
```

**Figure 2. Partitioned DFG**

The dataflow graph for a block of code has two types of vertices that do not represent instructions. *Sources* represent the state at the beginning of the block (no incoming edges) and *sinks* represent the state at the end of the block (no outgoing edges). One source and one sink exist for every register. Two special sinks exist for the PC and memory. The sink for the PC is required because otherwise an error in a branch or jump (e.g., "`bnez r1, target`" being decoded as "`bnez r3, target`") would have no impact on the dataflow graph and would thus be undetectable. The sink for memory is required to capture the effects of stores, and the output edges of all stores in a block flow into this memory sink. We maintain only a single sink for all of memory, because it is infeasible to maintain a separate sink for every possible location in memory, and dataflow in memory cannot be statically determined. Because of this simplification, DDFV will only ensure that store values reach memory correctly, but not that subsequent loads to that address will read the value written by the most recent store.

When a block is executed, data values flow along the edges from the sources to the sinks. Every value flowing out of a vertex has a unique history of vertices involved in its creation. We refer to the set of histories flowing into a vertex as the *input history* of the vertex, and we refer to the history of values flowing out of a vertex as the vertex's *output history*. A vertex's output history is defined recursively as a combination of the histories of the inputs to that vertex and information about the vertex (instruction type, immediates, etc.). The size of a vertex's input or output history depends on the length of the dependency path and is unbounded, which complicates storage and computation. To overcome this problem, DDFV uses a fixed-size checksum of the vertex's history, called the *vertex history signature (VHS),* instead of the full history. A checksum over the input histories of all sink vertices (i.e., histories of values flowing into sinks) represents the block's full dataflow graph and is called the *dataflow graph signature (DGS)*.

The functions that compute the output VHS of each vertex differ slightly for each vertex type, as we discuss in Section 5.1. The output history for each source is a constant, referred to as the *initial history*, that is unique to the source. When choosing the actual hash functions for computing the DGS and the VHSs, we must ensure that they are simple to implement in hardware and that they minimize the probability of *aliasing* (i.e., two different dataflow histories mapping to the same signature). However, with finite-sized signatures, there is always a non-zero probability of aliasing and thus *false negatives* (undetected errors). Therefore, DDFV can detect all errors within its coverage area, but only with a certain probability dependent on the number of bits in the signatures.

### 4.2. Providing Static DGSs to Hardware

Before a program is executed, we must identify code blocks with static dataflow graphs, compute the DGS for each such block, and store the DGSs in a way that makes it easy to locate and retrieve them at runtime. In this paper, we compute a DGS for every basic block. We have written a binary analysis/rewriting tool that identifies basic blocks using symbol and relocation information in the binary. For each basic block, the dataflow graph is reconstructed to compute the VHSs for all sinks and the DGS. This step could alternatively be performed by the compiler backend or the JIT compiler in dynamically compiling virtual machines.

Once the DGSs for all blocks are known, they have to be made accessible to the processor at runtime. To avoid the necessity of an additional storage structure, we embed the static DGS value in the program binary by making the first instruction of every basic block a *DGS instruction* that contains the signature in its immediate field. To implement the DGS instruction, we define a new opcode and modify the decoder to recognize it and extract the signature. This process of

embedding instructions in a binary is similar to that used in prior work [17].

### 4.3. Runtime Operation

At runtime, DDFV hardware computes the DGS of each basic block it executes by computing output histories for all instructions executed and tracking input histories for all registers, PC, and memory. We discuss the details of our particular implementation in Section 5.2.

As described in Section 4.2, the static DGS for every block is embedded in the program code as a special DGS instruction. Beside providing the actual signature, the DGS instruction also lets the processor know when to compare the static and dynamic DGS and reset the (dynamic) DGS and VHSs. When a DGS instruction is ready to commit, all instructions in the previous block must have already committed and updated the DGS register. Therefore the DGS register reflects the final signature of the previous block. At this point, the dynamically computed DGS is compared to the static DGS in the previous DGS instruction.

Like other instructions, DGS instructions can be squashed in case of branch mis-prediction and need to be tracked while speculative. There are two design options that we considered for tracking in-flight DGS instructions: we can either dispatch them to the ROB or into a separate FIFO that is dedicated to DGS instructions. Unlike ROB entries for instructions with output values that are often read multiple times by later instructions, entries for DGS instructions are only read during the commit stage. Therefore they can easily be stored in a separate DGS instruction FIFO, which does not require multiple read ports and is narrower than the ROB, because each entry only needs to be as wide as a DGS. We assume a separate FIFO in our implementation, because it reduces pressure on the ROB and therefore minimizes DDFV's performance impact.

## 5. Implementation Details

In this section, we provide the details of our DDFV implementation.

### 5.1. Signature Computation Functions

There are two types of dataflow graph vertices for which we compute a VHS: instruction outputs and sinks. We also compute the DGS for each basic block.

**Instruction Output VHSs.** The VHS of an instruction output, $VHS_{inst}$, depends on the type of instruction. In general, $VHS_{inst}$ is computed as a hash of the VHSs of its inputs, the immediate operand (if any), and an identifier of the operation performed. We use CRC as the hash function.

**Sink VHSs.** There are three types of sinks: registers, PC, and memory. The VHS of a sink refers to the sink's *input* history, unlike the VHS of an instruction (which refers to the instruction's output history).

The VHS of a register sink, $VHS_{reg}$, is simply the VHS of the last instruction output to write that register.

The VHS of the PC sink, $VHS_{pc}$, is updated after every branch instruction. In this work, there is only one branch per dataflow graph but, in general, the PC sink history can be determined by multiple input histories (if the DGS is computed over multiple basic blocks). In the case of multiple branches, DDFV uses a combining function, $comb_{pc}$, to summarize them. For ease of implementation, we require that $VHS_{pc}$ can be computed incrementally. In our implementation, the combining function is base-1 addition.

$$VHS_{pc,new} = comb_{pc}(VHS_{pc,old}, VHS_{inst,branch})$$

The history for the memory sink, $VHS_{mem}$, is computed using a combining function, $comb_{mem}$, that summarizes all of the store output histories. As with $VHS_{pc}$, we use base-1 addition for this combining function. $VHS_{mem}$ is seeded with a constant initial value at the beginning of each block, and it is updated whenever a new store commits. $VHS_{mem}$ covers the entire memory and can verify dataflow from registers to memory, but it cannot verify that stored data reaches future loads correctly.

$$VHS_{mem,new} = comb_{mem}(VHS_{mem,old}, VHS_{inst,store})$$

**DGS.** The DGS is computed by hashing together all of the sink VHSs with XOR. To avoid two identical incorrect histories cancelling each other out and to detect register sinks with swapped histories (despite using the commutative XOR function), we first hash each $VHS_{reg}$ using a function that depends on the register number. This hash function is a table-driven permutation function.

$$DGS = hash_{reg}(1, VHS_{reg1}) \oplus ... \oplus hash_{reg}(N, VHS_{regN}) \oplus VHS_{mem} \oplus VHS_{pc}$$

Figure 3 illustrates an example of DDFV detecting an error in the same basic block as shown in Figure 2. For clarity, we simplified the system in several ways. It has only 5 registers, the DGS and VHSs are 4-bits long, and hash functions and initial values for the DGS and VHSs are simplistic.

### 5.2. Dynamic DGS Computation

As the processor is executing, it must track the histories of values produced by the instructions (i.e., vertex

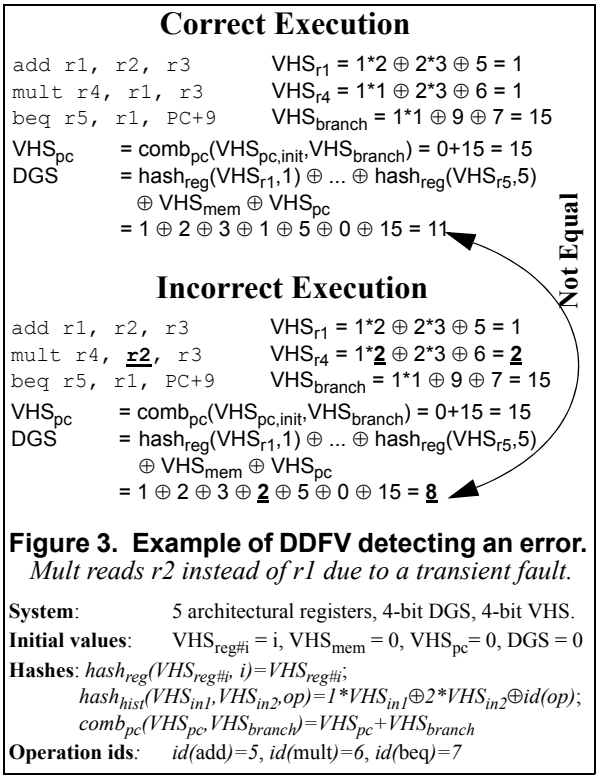



**Figure 3. Example of DDFV detecting an error.**
*Mult reads r2 instead of r1 due to a transient fault.*

**System**: 5 architectural registers, 4-bit DGS, 4-bit VHS.
**Initial values**: $VHS_{reg\#i} = i$, $VHS_{mem} = 0$, $VHS_{pc} = 0$, DGS = 0
**Hashes**: $hash_{reg}(VHS_{reg\#i}, i) = VHS_{reg\#i}$;
$hash_{hist}(VHS_{in1}, VHS_{in2}, op) = 1*VHS_{in1} \oplus 2*VHS_{in2} \oplus id(op)$;
$comb_{pc}(VHS_{pc}, VHS_{branch}) = VHS_{pc} + VHS_{branch}$
**Operation ids**: $id($add$) = 5$, $id($mult$) = 6$, $id($beq$) = 7$

input and output histories in the dataflow graph) so it can compute the DGS for comparison with the signatures contained in the static binary. Initial signature values are held in a ROM. Throughout the section we assume PentiumPro-style, implicit register renaming (i.e., in-flight operands are tagged with ROB entry numbers). We discuss other system models in Section 5.6.

**5.2.1. VHS computation.** The VHSs propagate through the processor along with the values produced by instructions. Thus, we add a VHS field to every architectural register, every ROB entry, and to every operand in the operand bypass network. We also add single instances of $VHS_{mem}$ and $VHS_{pc}$ registers that are updated during the commit stage.

By keeping the VHSs in the register file and the ROB, instead of in separate structures, we avoid the need for extra decoders and we allow DDFV to detect errors in the wordline decoders of the register file (because an error will cause the wrong history to be read). Hardware necessary for maintaining the VHSs includes additional SRAM cells, bitlines, and sense-amps, but no new read or write ports, because VHSs and data values in a ROB entry or register are always accessed together.

Histories are processed analogously to the data values to which they belong. Input operand histories are either read along with the data value during register fetch or received over the bypass network before execution. An instruction's new output history is computed during the execute stage using the equations from Section 4.1, and it is written to the ROB with the instruction output during writeback. Finally, when an ALU or load instruction commits, it updates its destination's VHS in the register file. Treating the history and data value as a unit makes an error in operand routing (e.g., incorrect renaming, scheduling, or bypassing) change the VHS of the instruction(s) consuming the operand and be detected. Stores and branches have no target VHS entries in the register file to overwrite; instead, they use the $comb_{mem}$ and $comb_{pc}$ functions to update $VHS_{mem}$ or $VHS_{pc}$.

**5.2.2. DGS computation.** At the end of every basic block, the DGS must be computed from the $VHS_{mem}$ and $VHS_{pc}$ registers and all the $VHS_{reg}$ values in the register file. Computing the DGS by summarizing all sinks at the end of every basic block would be difficult, because it would require reading the $VHS_{reg}$ field from every single register. Instead, we maintain an intermediate DGS that always represents the summary of the current $VHS_{reg}$ values and is updated whenever one of the registers is written during commit. Changes to $VHS_{mem}$ and $VHS_{pc}$ are not immediately reflected in the DGS, because the changes require read-modify-write updates using the combining functions. Instead they are XORed with the intermediate DGS at the end of every basic block to obtain the final DGS.

To update the DGS whenever a $VHS_{reg}$ value is updated, we must replace the register's old history ($VHS_{reg\#i}$) with its updated history ($VHS'_{reg\#i}$) in the equation used to compute the DGS. The DGS is an XOR over the hashed $VHS_{reg}$ values (see Section 5.1) and allows for simple updating because of the special properties of the XOR operator (commutative, associative, A ⊕ A = 0, and A ⊕ 0 = A). To replace $hash_{reg}(VHS_{reg\#i}, i)$ with $hash_{reg}(VHS'_{reg\#i}, i)$, it is sufficient to XOR the DGS with $hash_{reg}(VHS_{reg\#i}, i) \oplus hash_{reg}(VHS'_{reg\#i}, i)$. Because $hash_{reg}(VHS_{reg\#i}, i)$ was already part of the DGS before the update and is XORed in again, it occurs twice in the XOR expression and the two occurrences cancel each other out.

Although the update itself is simple, it still requires us to extend the instruction commit process from one cycle to two cycles. In cycle one, we read the destination's original $VHS_{reg}$ from the register file and hash it with $hash_{reg}$. In cycle two, we update the DGS using the old and the new destination $VHS_{reg}$, and we write the updated destination $VHS_{reg}$ from the ROB to the register file. This way of updating the DGS appears to require an extra read port on the register file. Instead,

we can convert the write port used to update the register values and histories into a two-cycle read-modify-write port with less extra hardware.[2]

**5.2.3. Resetting the DGS and VHSs.** After each basic block, we must reset the DGS and all VHSs to their initial values because they now represent source vertices in the dataflow graph. Resetting $VHS_{mem}$, $VHS_{pc}$, and the DGS is straightforward, because they are updated in-order at instruction commit, but resetting the $VHS_{reg}$ values to their initial values is more challenging. An instruction reading the output $VHS_{reg}$ can belong to either the same basic block or a subsequent basic block. Instructions in the same basic block expect the computed output history, but instructions in later basic blocks expect the $VHS_{reg}$ to be reset to its initial value. Due to reordering, instructions from the same basic block can read the $VHS_{reg}$ before or after instructions from subsequent basic blocks. Hence there is no point in time when all $VHS_{reg}$ values can be reset.

Thus, in our implementation, we actually have two VHS fields in each ROB entry. Only ALU and load instructions use both fields to hold two values: $VHS_{reg,ex}$ is the VHS of the destination register, as computed during the execute stage, and $VHS_{reg,init}$ is the initial VHS of the destination register (i.e., assuming the register is a source vertex). Intuitively, $VHS_{reg,ex}$ will be used by consumer instructions in the same basic block, and $VHS_{reg,init}$ will be used by consumers in future basic blocks. As only one value is provided for all future blocks, some errors that cause a value to be forwarded to the wrong future block can go undetected. This problem could be mitigated by cycling through multiple sets of initial values, but this solution would require one DGS to be embedded for each set of initial values. In systems where the number of inputs and thus $VHS_{reg,init}$ values is smaller than the number of possible history values, $VHS_{reg,init}$ can be replaced with a shorter $VHS_{reg,seed}$ used to compute or look up the initial value.

Besides the two VHS fields, this resetting scheme requires 3 control bits per ROB entry to determine if the operation is the first writer of its destination register within the current basic block (*FirstWriterThisBB*) and if either input value is passed from the previous basic block (*IsInput[A/B]FromPrevBB*). These three bits are computed after decode (in rename or dispatch), when operations are still in order and architectural registers are already known, using a bitmask of all registers that have not been used as destinations within the current basic block.

These modifications affect DDFV's operation as described thus far. During register read, an instruction uses the two IsInputXFromPrevBB bits to decide if it should request, for each operand, the operand's current VHS value or initial VHS value. When an ALU or load instruction completes, it writes its destination value and both $VHS_{reg,ex}$ and $VHS_{reg,init}$ to its ROB entry. At commit, an ALU or load instruction writes its destination value and $VHS_{reg,ex}$ into the register file. Also at commit, an ALU or load instruction updates the DGS. The instruction uses its FirstWriterThisBB bit to determine whether to replace $VHS_{reg,ex}$ or $VHS_{reg,init}$ from the DGS before updating it.

## 5.3. Data Value Checking

As discussed earlier, the DGS only serves to verify the correct *shape* of the dataflow graph. To fully check correct dataflow, DDFV must further detect errors in data values flowing between vertices in the graph (i.e., instructions). For this purpose, each VHS is accompanied by a checksum (parity bit) of its corresponding data value. The checksum is stored alongside the VHS in the register file and ROB. When a new history is computed, the data checksum for the inputs is checked and a checksum for the output is computed.

## 5.4. Exceptions and Interrupts

Exceptions and interrupts violate our assumption that we statically know the expected dataflow graph between when a core enters execution of a basic block and when it exits. There are at least two solutions for adapting DDFV to handle exceptions and interrupts. First, we could simply choose to not check signatures of blocks that are not executed without interruption. If exceptions and interrupts are sufficiently rare, the impact on error detection coverage will be small. Second, we could make DDFV state visible to the operating system, such that it can be saved and restored along with other program state. The state that needs to be saved is small (~200-400 bits depending on the configuration). Our preference is to add hardware support for this second option but fall back to the first option for DDFV-oblivious operating systems.

## 5.5. Implementation Cost and Complexity

DDFV requires additions to various structures (summarized in Table 2) and slightly modifies the behavior of multiple pipeline stages (shown in Table 3). Never-

2. We add an extra set of bitlines, sense amps, and wordlines driven by the decoder of the write port to the SRAM cells storing the $VHS_{reg}$. We also add latches between the decoder outputs and the original wordlines.

**Table 2. DDFV Hardware Additions**

| Component | Hardware Additions |
|---|---|
| register file | $VHS_{reg,ex}$ field and EDC bit in each register<br>Read-modify-write port to replace write-port |
| reorder buffer | 2 VHS fields ($VHS_{reg}$,$VHS_{reg,init/seed}$)<br>3 control bits per ROB entry<br>1 bit EDC (parity) |
| execution units | Logic to compute updated DGS and VHSs<br>Logic to check and update EDC |
| back-end | DGS register, $VHS_{mem}$ and $VHS_{pc}$ registers<br>FIFO for in-flight signatures<br>Comparator to check dynamic DGS<br>Lookup table for initial DGS and VHS values |

**Table 3. DDFV Pipeline Behavior Changes**

| Stage | Behavior Modifications |
|---|---|
| fetch | *none* |
| decode | Put signatures from DGS instructions into FIFO |
| rename | *none* |
| dispatch | Compute 3 control bits in ROB entry. |
| reg. read | Read $VHS_{reg,ex}$ or $VHS_{reg,init}$ for input registers. |
| execute | Compute updated VHS for destination.<br>Check source EDC and compute new EDC. |
| complete | If destination is register, write $VHS_{reg,ex}$ and $VHS_{reg,init}$ into ROB entry. |
| commit | Now 2-cycles instead of one cycle.<br>If destination is register, copy $VHS_{reg,ex}$ to register file and update dynamic DGS.<br>If last instruction in basic block, finalize dynamic DGS and compare static DGS in FIFO. |

theless, neither the design nor the overall hardware cost of DDFV is prohibitive because all added fields are small and the required logic is simple. DDFV may appear complex compared to replication, but it is straightforward to add to an existing design. The biggest change—the addition of history fields to the ROB, registers, and bypass network—requires widening of existing structures and paths, but does not introduce new structures or datapaths. The VHS computation units are simple combinatorial circuits and identical throughout the processor. The logic to compute the DGS is a simple logic block in the commit stage.

The major area cost is incurred by the DGS FIFO as well as adding the VHSs to the ROB and register file. In sum these add roughly 0.2 $mm^2$ to the processor configuration in Table 4 implemented in 130nm, as estimated using Cacti [13]. This does not include the area for the wider bypass network and other data buses, which is impossible to assess without an actual design. The area cost of the logic for VHS updates, DGS computation, and parity generation is negligible.

Although adding hardware always increases the potential for hardware faults, errors in DDFV hardware cannot impact program correctness. A transient error in the DGS computation can only lead to *false positives* (errors signalled despite correct execution) and unnecessary error recoveries. Permanent errors in the DDFV logic could prevent forward progress by continuously signalling errors, but this condition can be detected using retry counters.

### 5.6. Other System Models

The implementation of DDFV needs to be adjusted to accommodate features in some microarchitectures. We discuss two such issues in this section.

**Explicit Renaming.** We have assumed a processor that renames registers using the ROB instead of an explicit map table. With explicit renaming, the register histories, $VHS_{reg}$, are all stored in the register file and not in the ROB. The *FirstWriterThisBB, IsInputAFromPrevBB,* and *IsInputBFromPrevBB* state bits remain in the ROB. $VHS_{mem}$, $VHS_{pc}$, and DGS still only have one instance each, do not participate in renaming, and are updated during the commit stage.

**Cracking of x86 instructions.** Converting macro-ops into micro-ops changes the shape of the dataflow graph. We can overcome this problem by ensuring that the output history for the macro-op dataflow graph equals that of the micro-op dataflow graph. At a high level, we must design the VHS update functions for the micro-ops such that their composition is equal to the VHS update function for their macro-op. We have developed a matrix-based hashing scheme for this purpose, and we describe it in Appendix A. Matrix-based hashing logic requires similar area as CRC for an equal number of VHS bits.

## 6. Evaluation

To evaluate our DDFV implementation, we need to determine the impact of aliasing on DDFV's error coverage (Section 6.1) and the performance penalty caused by the embedded DGS instructions (Section 6.2). We use SimpleScalar [4] to simulate the processor model described in Table 4. We sample from the SPEC 2000 benchmark suite using SimPoints [26].

### 6.1. Error Coverage

There are two aspects to DDFV's error coverage: the chip area in which DDFV can potentially detect errors and the number of errors that go undetected due to aliasing. Simulation at the micro-architectural level is not suited to evaluate the former aspect, but it does provide insights into the latter. Faults within DDFV's cov-

erage area manifest themselves as errors on the micro-architectural level and the underlying physical fault is irrelevant for aliasing.

To determine the impact of aliasing, we injected single-bit errors (10000 per benchmark) and observed whether DDFV detected them. The errors included: errors in fetched instructions; incorrectly decoded register numbers, opcodes, and immediates; errors in register renaming; wrong register accessed during register read; and wrong register updated in writeback or commit. We did not include single-bit errors injected into data values, because these are known to be detected using EDC. Many of the injected errors have no effect on correct program execution because of masking (e.g., faulty instruction squashed after misspeculation, error in unused instruction bit, etc.). Masked errors (47% of injected errors) are not considered in our results.

In Figure 4, we plot the fraction of injected errors not detected as a function of the output history size in bits, which is the same for $VHS_{reg}$, $VHS_{branch}$, and $VHS_{store}$. Whereas the width of these output histories determines the size of the VHS field added to every entry in the register file and ROB, the width of the summary histories (DGS, $VHS_{mem}$, and $VHS_{pc}$) only affects the three registers added to store them and making them wider is inexpensive. Thus the width of these histories will be determined by the number of bits available for the signature in the DGS instruction. We assume a 24-bit signature in each DGS instruction (32-bit instruction minus 8-bit opcode). The results are aggregated across all benchmarks, and they show that the fraction of undetected errors decreases exponentially as the number of history bits increases.

**Table 4. Processor Configuration**

| | |
|---|---|
| pipe depth | 4 front-end stages, variable execute, 1 commit stage (2 with DDFV) |
| pipe width | 4-wide decode/issue/commit |
| reorder buffer | 128 entries |
| ld-st queue | 32 entries |
| integer FUs | 4 ALUs, 1 integer multiplier |
| fl. pt. FUs | 4 FPUs, 1 FP multiplier |
| branch pred. | GShare, 32Kbits, 15-bit history |
| L1I cache | 32KB, 2-way, 2-cycle hit, 32B blocks |
| L1D cache | 32KB, 4-way, 2-cycle hit, 32B blocks |
| L2 cache | 1 MB, 8-way, 8-cycle hit, 64B blocks |
| memory | 48 cycles latency until first 8-bytes, 55 cycles latency until last 8-bytes |
| VHS/DGS | 10-bit VHS for insts and regs; 24-bit DGS, $VHS_{mem}$, and $VHS_{pc}$; 5-bit $VHS_{reg,seed}$; |
| DGS FIFO | 64 entries |

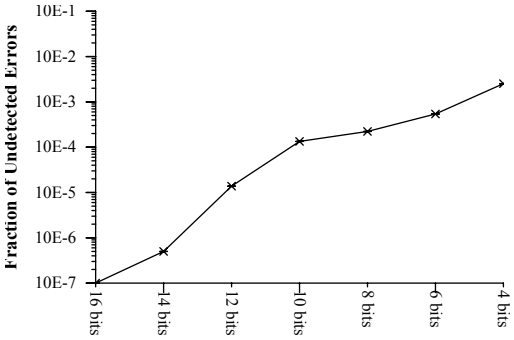



**Figure 4. Fraction of undetected errors as a function of $VHS_{reg/store/branch}$ size (in bits)**

Assuming uniform fault-sensitivity across the chip, DDFV's FIT reduction is equal to coverage area multiplied by the error detection probability. Based on die schematics of various processors—AMD K8 (see Figure 1), MIPS R10000, Pentium 4, and Alpha 21264 (not shown)—we estimate DDFV's coverage to be around 35%-50% of the fault-sensitive core area. Thus, FIT reduction is dominated by coverage area, as the detection probability is over 99.5% even for 6-bit histories. Estimates from die areas are not very accurate and vulnerability is likely to be non-uniform, but the estimates indicate that DDFV covers a significant portion of the core. More accurate estimates would require error injection experiments for a DDFV implementation in a circuit-level model of a commercial superscalar processor, which is unobtainable outside industry.

## 6.2. Performance

DDFV can degrade performance because of the embedded DGS instructions which consume fetch, decode, and commit bandwidth as well as instruction cache entries. DDFV also adds a cycle to commit, which adds pressure to the ROB and LSQ. An independent performance issue could arise from widening the ROB and the register file if either one is critical to timing. However, access latency for these structures is dominated by the (unmodified) word-line decoder, rather than word width [3]. Thus, we do not consider changes in clock cycle times in our experiments.

In Figure 5 we show the experimental results of comparing the performances of a processor with DDFV and an unprotected processor. We assume that DGS instructions do not consume ROB entries (i.e., they are stored in a dedicated FIFO structure). The results show that DDFV has limited effect on average performance with a mean slowdown of 1.8% in Figure 5, but it varies considerably between benchmarks. Some benchmarks with large basic blocks experience no slowdown, whereas the slowdowns for branch-heavy code with small basic blocks (e.g., *gcc*, *perl*, and *crafty*) approach 5%. *apsi* experiences a neg-

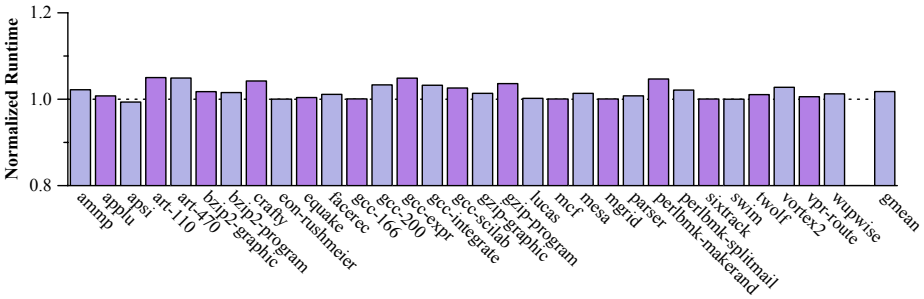



**Figure 5. Runtime with DDFV relative to unprotected baseline processor.**

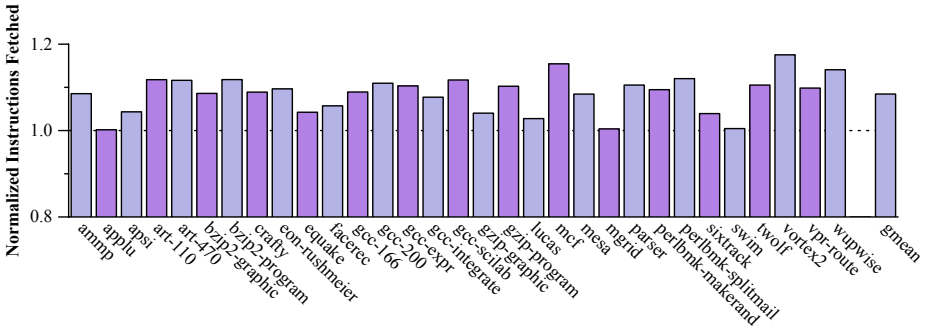



**Figure 6. Instructions fetched with DDFV relative to unprotected baseline processor.**

ligible speedup, because code rewriting leads to a realignment that reduces conflict misses in the L1I. We could potentially improve performance for branch-heavy code by computing signatures over paths rather than basic blocks [6].

Figure 6 shows the number of instructions fetched, normalized to a processor without DDFV. The processor with DDFV fetches on average 8.5% more instructions from the L1I cache to obtain the embedded signatures. DDFV's slowdown (1.8%) is less than the increase in the number of instructions fetched (8.5%), because many benchmarks are either not limited by the fetch stage at all or limited by the processor's inability to fetch across a taken branch (within a cycle) rather than fetch width.

## 7. DDFV Usage Models

DDFV can be used on its own or in conjunction with previously developed error detection schemes. By itself, DDFV protects a large number of components at very low cost. Some of the protected units, like the instruction fetch queue and the RAT, are known to be highly vulnerable to faults [20]. Unlike parity bits, DDFV not only protects the data stored in these structures and latches, but also the associated access logic.

DDFV can be inexpensively combined with control flow checking by changing the semantics of the DGS instruction. If, instead of the signature for the current block, it stores the signature for the successor block, inter-block control flow will be checked implicitly. Special considerations are necessary for conditional branches (two successors) and indirect branches (successor specified in register). Details are beyond the scope of this paper.

DDFV can reduce the cost of DMR schemes by protecting the processor front-end without hardware replication or reduced performance. With DDFV protecting the early pipeline stages, DIVA no longer has to replicate them in the checker core. This is particularly helpful for processors with little speculation or difficult-to-simplify fetch and decode logic. With DDFV and an EDC-protected register file, DIVA no longer needs a redundant register file.

DDFV can alleviate RMT's pressure on the fetch and decode stages, which are typically a bottleneck in SMT and RMT processors [30]. As described by Kumar et al. [16], performance overhead can be reduced by replicating instructions after they have been fetched and decoded. Kumar et al. propose to protect the pipeline front-end using parity bits, which can protect storage structures but not logic. With DDFV instead of parity, we provide full protection of the instructions before replication and detect permanent errors in rename, scheduling, and writeback stages that elude RMT.

## 8. Conclusions

DDFV offers a high-performance, low-cost solution for comprehensively detecting transient faults, perma-

nent faults, and design bugs in a large fraction of the microprocessor core. It is not a replacement for all existing error detection mechanisms, but instead can be used to provide inexpensive partial coverage or to reduce the cost of full dynamic verification schemes such as RMT or DIVA. Given the increasing need to detect errors, we expect dynamic verification approaches like DDFV to become increasingly attractive to architects. The comprehensive nature of dynamic verification provides coverage for a large portion of the processor rather than requiring low-level, component-specific error detection mechanisms. Beyond today's superscalar processors, DDFV is also a good fit for emerging architectures with distributed computation such as TRIPS [21].

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CCR-0309164 and EIA-9972879, the National Aeronautics and Space Administration under grant NNG04GQ06G, an equipment donation from Intel Corporation, and a Duke Warren Faculty Scholarship. We thank Alvy Lebeck, Chris Dwyer, Mike Bauer, Fred Bower, Curt Harting, Anita Lungu, Bogdan Romanescu, Amir Roth, and David Wood, for helpful discussions about this work.

## References

[1] Advanced Micro Devices. AMD Eighth-Generation Processor Architecture. Advanced Micro Devices Whitepaper, Oct 2001.

[2] Advanced Micro Devices. Revision Guide for AMD Athlon64 and AMD Opteron Processors. Publication 25759, Revision 3.59, Sept. 2006.

[3] B. S. Amrutur and M. A. Horowitz. Fast Low-Power Decoders for RAMs. *IEEE Journal of Solid-State Circuits*, 36(10):1506–1515, Oct 2001.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[5] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual Int'l Symp. on Microarchitecture*, pp. 196–207, Nov. 1999.

[6] T. Ball and J. Larus. Using Paths to Measure, Explain, and Enhance Program Behavior. *IEEE Computer*, 33(7):57–65, July 2000.

[7] X. Delord and G. Saucier. Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors. In *Proceedings of International Test Conference*, pp. 936–945, 1991.

[8] G. Hinton et al. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Feb. 2001.

[9] D. Hunt and P. Marinos. A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems*, pp. 170–175, 1987.

[10] T. I. Inc. TMS320C54x DSP Reference Set, Mar. 2001.

[11] Intel Corporation. Intel Pentium 4 Processor Specification Update. Document Number 249199-065, June 2006.

[12] International Technology Roadmap for Semiconductors, 2003.

[13] N. P. Jouppi and S. J. Wilton. An Enhanced Access and Cycle Time Model for On-Chip Caches. DEC WRL Research Report 93/5, July 1994.

[14] S. Kim and A. K. Somani. On-Line Integrity Monitoring of Microprocessor Control Logic. In *Proc. of the Int'l Conf. on Computer Design*, pp. 314–319, Sept. 2001.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[16] S. Kumar and A. Aggarwal. Self-checking Instructions: Reducing Instruction Redundancy for Concurrent Error Detection. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, pp. 64–73, Sept. 2006.

[17] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pp. 291–300, June 1995.

[18] C. McNairy and D. Soltis. Itanium 2 Processor Microarchitecture. *IEEE Micro*, 23(2):44–55, Mar/Apr 2003.

[19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *29th Annual Int'l Symp. on Computer Architecture*, pp. 99–110, May 2002.

[20] S. S. Mukherjee et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. of the 36th Annual Int'l Symp. on Microarchitecture*, Dec. 2003.

[21] R. Nagarajan et al. A Design Space Evaluation of Grid Processor Architectures. In *Proc. 34rd Annual Int'l Symp. on Microarchitecture*, pp. 40–51, Dec. 2001.

[22] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–74, Mar. 2002.

[23] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proc. 27th Annual Int'l Symp. on Computer Architecture*, pp. 25–36, June 2000.

[24] G. A. Reis et al. SWIFT: Software Implemented Fault Tolerance. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, pp. 243–254, Mar. 2005.

[25] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. 29th Int'l Symp. on Fault-Tolerant Computing Systems*, pp. 84–91, June 1999.

[26] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[27] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on*

*Dependable Systems and Networks*, June 2002.

[28] S. Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proc. of the Twelfth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[29] J. Srinivasan et al. The Impact of Technology Scaling on Lifetime Reliability. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, June 2004.

[30] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 191–202, May 1996.

[31] N. J. Warter and W.-M. W. Hwu. A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking. In *Proc. 20th Int'l Symp. on Fault-Tolerant Computing Systems*, pp. 442–449, June 1990.

[32] C. Weaver and T. Austin. A Fault Tolerant Approach to Microprocessor Design. In *Int'l Conf. on Dependable Systems and Networks*, pp. 411–420, July 2001.

## Appendix A: Handling Micro-Ops

We can adapt DDFV to processors that break instructions into micro-ops before execution. Such processors (e.g., AMD Athlon [1] and Intel Pentium 4 [8]) are a particularly interesting target for DDFV because they are expensive to replicate and cannot easily be simplified to fit into a DIVA-like scheme. The conversion into micro-ops changes the shape of the dataflow graph as some of the original nodes (macro-ops) are replaced by a sub-graph (micro-ops) that differs between vendors and processor generations.

Figure 7 illustrates different implementations of the same CISC *add* instruction, which exhibits two features found in many CISC ISAs: memory operands and complex addressing modes. To make DDFV portable across all implementations without changing the static signatures, the output histories of all dataflow graphs must be identical. In the native CISC implementation, the output VHS is a simple hash of the operation and

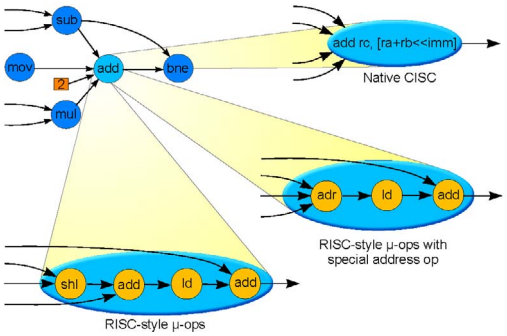



**Figure 7. Various implementations of a CISC instruction**

**Native CISC (2-operand format)**

`add rC, MEM[rA+rB<<i]` $VHS_{rC} \Leftarrow [1,2,6,4,5] \times [2,5,6,2,1]^T = \mathbf{1}$

**RISC-style μ-ops w/ special address op**

`adr tmp, [rA+rB<<i]` $VHS_{tmp} \Leftarrow [1,8,2,7] \times [5,6,2,1]^T = 4$
`ld tmp, [tmp]` $VHS_{tmp} \Leftarrow [8,6] \times [4,1]^T = 8$
`add rC, tmp, rC` $VHS_{rC} \Leftarrow [4,1,7] \times [8,2,1]^T = \mathbf{1}$

**RISC-style μ-ops**

`shl tmp, rB, imm` $VHS_{tmp} \Leftarrow [2,3,5] \times [6,2,1]^T = 3$
`add tmp, tmp, rA` $VHS_{tmp} \Leftarrow [4,1,7] \times [3,5,1]^T = 4$
`ld tmp, [tmp]` $VHS_{tmp} \Leftarrow [8,6] \times [4,1]^T = 8$
`add rC, tmp, rC` $VHS_{rC} \Leftarrow [4,1,7] \times [8,2,1]^T = \mathbf{1}$

**Figure 8. Example of VHS computation for μ-ops.**
*Initial histories rA=5, rB=6, rC=2, i(mmediate)=2. All operations performed in modulo-10 arithmetic.*
*Left multiplicands are representative matrices (e.g. [4,1,7] = add regs), the right ones are input history vectors.*

inputs. In the micro-op case, the resulting history is a hash over hashes from earlier micro-ops. One way to enforce that the final output histories are equal is to use matrix multiplications as hash functions. In this scheme, each operation (micro-op or macro-op) is assigned a unique representative matrix that has one row for each output of the operation and one column for each input plus an additional column to avoid mapping zero-vectors to zeroes (e.g., an addition would be represented by a 1x3 matrix). Output history values are computed by multiplying the representative matrix with an input history vector containing all input history values and a constant one (see Figure 8).

The easiest way to develop representative matrices for macro-ops is bottom-up. The macro-ops are first split into the smallest possible set of tasks (even if the micro-ops used in the implementation perform multiple of these tasks). Next all the tasks are assigned representative matrices. The macro-op matrix as well as the matrices for the micro-ops are derived from the task matrices by computing output histories symbolically. Representative matrices can also be developed top-down (i.e., starting with a macro-op matrix) if tasks become smaller than anticipated.

The simplistic example in Figure 8 uses modulo-10 arithmetic, but such matrix multiplications would be prohibitively expensive. Instead we interpret the input and output histories as bit vectors rather than numerical values and multiply them with binary matrices, which is inexpensive as +=*xor* and *=*and*. This scheme works only for simple macro-ops that break into a constant number of micro-ops. Uncommon, more complex macro-ops emit pseudo-histories for their outputs upon completion.