Dataflow Dominance: A Definition and Characterization

Matt Ramsay

University of Wisconsin-Madison
Department of Electrical and Computer Engineering
1415 Engineering Drive
Madison, WI 53706

Submitted in partial fulfillment of the M.S. Degree in Electrical and Computer Engineering*
Project Option

December 2003.

Dataflow Dominance: A Definition and Characterization

Abstract

In this work, we define and characterize a new program attribute called dataflow dominance. We find that there is a non-trivial percentage of 2-input instructions whose results can be determined by the computation and fetching of only one operand (example: mult rd, rs, rt=0). We call these instructions dataflow dominant instructions. We show that performance can be gained not only from allowing the dataflow dominant instruction to execute early, but more significantly from potentially eliminating the execution of all instructions that lead to the computation of the dominated operand. We find that, for our benchmarks, 0.1% - 6.5% (2% average) of the total instructions executed contain a dataflow dominant operand and that 0.6% - 23.3% (8% average) of the remaining instructions can be eliminated with an optimal prediction and detection mechanism.

We show that the occurrence of these dataflow dominant operands, as well as the instructions that can potentially be eliminated by their arrival, is a repeatable trend. We therefore propose two speculative mechanisms to dynamically remove instructions deemed unnecessary by the arrival of a dataflow dominant operand. We find that removing these instructions not only results in an overall performance improvement, but also improves cache behavior and allows for a significant increase in the instructions window size available for the scheduling and execution of useful instructions.

1 .0 Introduction and Motivation

Despite the efforts of superscalar and multi-threaded processors to increase both the IPC and throughput that can be obtained from today's workloads, the fundamental limit of a program's performance still lies primarily within the dataflow graph made by the true dependences contained in the code. Put more simply, a program can only finish as quickly as its critical path of dependent instructions executes. In order to push performance past this limit, hardware must be able to either break data dependences by speculating on data values [7], by removing instructions from the dependence chain, or both, all without affecting program correctness.

In this work, we define and characterize a program attribute called dataflow dominance. We show that with realistic hardware exploitation this phenomenon allows for the elimination of a non-trivial percentage of instructions that do not contribute to the end result of the program. We find that removing these unnecessary instructions not only results in an overall performance improvement, but also improves cache behavior and allows for a significant increase in the instructions window size available for the scheduling and execution of useful instructions.

We show that there is a non-trivial percentage of 2-input instructions whose results can be determined by the computation and fetching of only one operand. We call these instructions *dataflow dominant*. The arrival of the dataflow dominant operand makes the value of the other operand irrelevant. Therefore, the instructions that compute the dominated operand are also unnecessary and may be removed from execution without affecting the program's overall result.

We show that an optimal exploitation of dataflow dominance leads to correct execution of our benchmarks while executing as many as 23.3% fewer instructions. This same optimal exploitation leads to as much as a 15.5% speedup of the total execution. To take advantage of this opportunity, we propose two mechanisms capable of dynamically removing unnecessary instructions from execution. The first is a predictor based mechanism that only eliminates the execution of unnecessary instructions, while the second, similar to a trace cache [9][10][13], allows the machine to fetch around predicted unnecessary instructions, thus saving further processor resources.

This paper is organized as follows: Section 2 presents related work, Section 3 defines and characterizes dataflow dominance, Section 4 discusses potential performance gains from its exploitation, Section 5 describes possible implementations, and Section 6 presents our conclusions.

2 .0 Related Work

There has been significant work aimed at removing the portions of programs that perform no useful work. Butts and Sohi proposed a mechanism to detect and dynamically remove dead instructions, instructions whose results are never read [4]. If an instruction's result is found to be overwritten before it is used as an input by a later instruction, it is declared dead. Since most values are only read by one later instruction [3], minimal state is needed to keep enough instructions in the machine to locate the redefinition of the dead instruction's output register. Butts and Sohi also defined a class of instructions as transitively dead. These instructions generate results only used by dead instructions or other transitively dead instructions. In other words, transitively dead instructions reside in a path of the dataflow graph that is terminated by the presence of a dead instruction. Dead and transitively dead instructions have a relationship similar to that of dataflow dominant instructions and the instructions that they make unnecessary, except that unlike dataflow dominant instructions, the dead instructions themselves can be removed. While they propose a mechanism for the removal of dead instructions, transitively dead instructions are not targeted for exploitation. Our mechanism (Section 5) targets dead and transitively dead instructions for removal along with instructions made unnecessary by the presence of a dataflow dominant instruction.

Zilles identified a set of instructions related to dataflow dominant instructions called idempotent instructions

[17]. An idempotent instruction produces an output identical to one of its inputs (ex: add rd, rs, rt=0). Idempotent instructions have the potential to be "executed" during the register rename stage and do not require the use of a functional unit. The exploitation of idempotent instructions has minimal benefit, however, because the values of both inputs are required for correct execution and therefore no benefit can be gained by the exploitation of the transitively idempotent property.

Other work in this area includes the following. Partial dead code elimination [5] is a compiler technique that attempts to reduce the number of dead instructions by placing all possible uses of a register value in the same control flow path. If this technique is successful, the number of dead instructions cannot be increased by irregular control flow. Rotenberg proposed exploiting (but not removing) ineffectual instructions [12], instructions that have no externally visible effects. These ineffectual instructions include dead and transitively dead instructions, silent stores [1][6], trivially predictable branches, and other instructions that transitively lead into another ineffectual instruction. His work exploited these instructions by removing them from speculative threads. Martin et al. proposed a compiler/hardware scheme to identify dead register values for the purpose of expediting physical register reclamation [8].

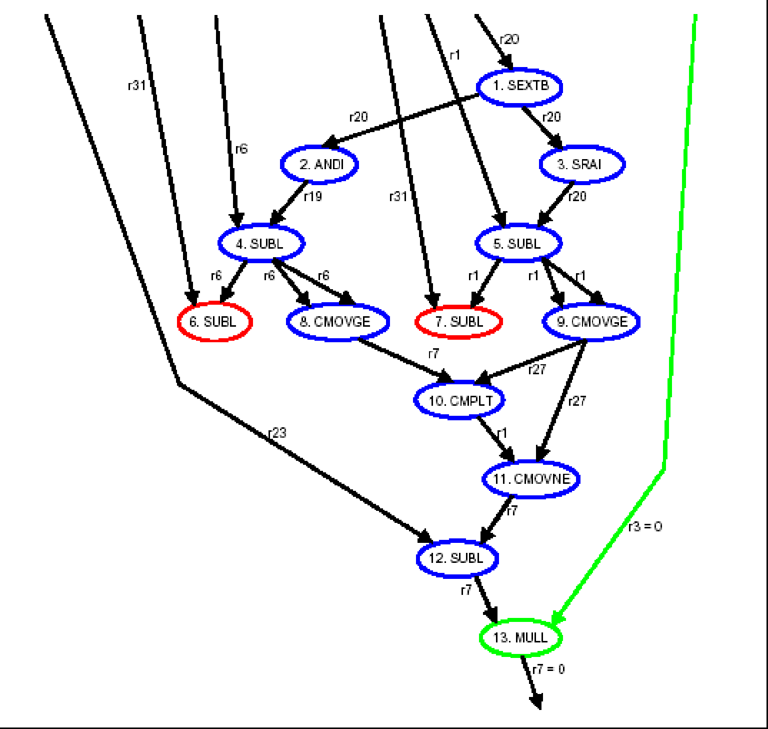
2.1 Floating-Point Idempotent Instructions

Our work also identifies a second group of idempotent instructions unique to floating-point instructions. An idempotent floating-point instruction occurs when the exponents of the two operands of a floating-point add or subtract instruction differ by more than the number of bits in the significand. When this condition is satisfied, the normalization performed on the operands before the floating-point add/subtract operation will completely shift out one of the operands, turning the instruction into an add/subtract by zero. After this normalization, floating-point idempotent instructions fall under the original definition of idempotent instructions coined by Zilles. Table 1 shows the percent-

Benchmark	Diff = 0	0 < Diff < 11	Diff > 52	Benchmark	Diff = 0	0 < Diff < 11	Diff > 52
ammp	8.6%	62.9%	0.2%	**lucas**	22.0%	71.3%	0.2%
apsi	33.3%	55.8%	0.6%	**mesa**	22.7%	58.3%	2.2%
art	0.8%	42.4%	8.3%	**mgrid**	15.3%	77.0%	0.6%
equake	17.8%	66.7%	3.9%	**sixtrack**	10.6%	63.2%	0.1%
facerec	14.7%	76.4%	1.9%	**swim**	34.0%	15.4%	1.6%
fma3d	32.0%	36.9%	14.6%	**wupwise**	16.9%	80.9%	0.0%
galgel	7.2%	63.4%	1.0%	**Average**	18.1%	59.3%	2.7%

Table 1: Floating-Point Exponent Difference: Shows the total percentage of floating-point add and subtract instructions whose exponents differ by the indicated margin. A difference of more than 52, the number of bits in the significand of a 64-bit floating-point value, indicates a floating-point idempotent instruction.

age of floating-point add/subtract instructions whose operands' exponents are the same, differ by less than ten, and differ by more than 52. Since there are 52 bits in the significand of a 64-bit floating-point value, those instructions whose operands' exponents differ by greater than 52 are floating-point idempotent and can be executed by a copy instruction instead of accessing the slower, more power-intensive floating-point unit. Table 1 also shows that these floating-point idempotent instructions occur very rarely (2.7% on average), especially considering that floating-point add/subtract instructions make up only 19% of the instructions in our benchmarks. For that reason, we chose not to target them in the remainder of our work. It is also apparent that the majority of floating-point add/subtract instructions (78.4% on average) have operands whose exponents differ by less than ten. This rules out the validity of bitwise optimizations, such as gating off power to the portions of the floating-point unit where the exponent normalization has made parts of the instruction idempotent on a bit-by-bit basis.



Code from Alpha ISA:
Format: INST rd, rs, rt

```
1. SEXTB r20, r20
2. ANDI r19, r20, IMM
3. SRAI r20, r20, IMM
4. SUBL r6, r6, r19
5. SUBL r1, r1, r20
6. SUBL r7, r31, r6
7. SUBL r27, r31, r1
8. CMOVGE r7, r6, r6
9. CMOVGE r27, r1, r1
10. CMPLT r1, r7, r27
11. CMOVNE r7, r1, r27
12. SUBL r7, r23, r7
13. MULL r7, r7, r3
```

Figure 1. Example of Dataflow Dominance from Crafty. The dataflow graph is generated from the code on the right. The dataflow dominant instruction and operand are shown in green (Instruction #13, r3=0), dead instructions in red (Instructions #6, 7), and additional instructions made unnecessary by dataflow dominance in blue (Instructions #1-5, 8-12). This dataflow dominant operand appeared 9234 of 9831 times in our 50M instruction trace.

3 .0 Dataflow Dominance

3.1 Dataflow Dominance Defined

Any instruction for which there exists one or more data values whose arrival as one of the instruction's input operands allows for its output to be known without examination of the other operand's value has the potential to be dataflow dominant. The instruction becomes dataflow dominant each time the dominant operand value appears on its input. In these cases, the value of the other operand, which we define as the *dominated operand*, need not be fetched or even computed for correct execution of the program, provided that its value is not needed as input to any other instruction(s).

The simplest example of this phenomenon is multiply-by-zero, where the arrival of a zero value on either of the operands automatically results in the output being zero regardless of the other operand's value. In this case, all instructions in the program whose only function is to compute the dominated operand can be removed without affecting program correctness. An example of this situation, taken from the crafty benchmark, is shown in Figure 1. In this example, the arrival of r3=0 as an input to Instruction #13 makes it a dataflow dominant instruction. This also: 1) designates the r7 input to Instruction #13 as the dominated operand and 2) makes Instructions #1 - #12 unnecessary for program correctness.

Instruction	Condition(s) of Dataflow Dominance	Instruction	Condition(s) of Dataflow Dominance
AND	rs == 0 or rt == 0	CMOVLE	rs > 0
BIC	rs == 0 or rt == oxFFFFFFFF	CMOVGT	rs <= 0
BIS	rs == 0xFFFFFFFF or rt == oxFFFFFFFF	CMOVLBS	(rs & ox1) == 0
ORNOT	rs == 0xFFFFFFFF or rt == 0	CMOVLBC	(rs & 0x1) == 1
ZAP	rs == 0 or (rt & 0xFF) == 0xFF	MULS	rs == 0 or rt == 0
ZAPNOT	rs == 0 or (rt & 0xFF) = 0	DIVS	rs == 0
SRL	rs == 0	MULT	rs == 0 or rt == 0
SLL	rs == 0	DIVT	rs == 0
SRA	rs == 0 or rs = 0xFFFFFFFF	CPYS	rt == 0
MULL	rs == 0 or rt == 0	CPYSN	rt == 0
MULQ	rs == 0 or rt == 0	FCMOVEQ	rs != 0
UMULH	rs == 0 or rt == 0	FCMOVNE	rs == 0
CMOVEQ	rs != 0	FCMOVLT	rs >= 0
CMOVNE	rs == 0	FCMOVGE	rs < 0
CMOVLT	rs >= 0	FCMOVLE	rs > 0
CMOVGE	rs >= 0	FCMOVGT	rs <= 0

Table 2: Dataflow Dominance Conditions.

3.2 Dataflow Dominance in the Alpha Instruction Set

The thirty-two instructions in the Alpha instruction set [14] that have the potential to be dataflow dominant are shown in Table 2. As stated earlier, all of these instructions are 2-input instructions, where meeting of the listed condition on one of its input registers dominates the value of the other input. Table 3 further characterizes the occurrence of dataflow dominant instructions. This data shows that the vast majority of dataflow dominance cases occur in only a few instructions (78.3% in 6 instructions for spec_INT, 93.7% in 6 instructions for spec_FP), and that these instructions are dominant a high percentage of the time. Table 3 also shows that dataflow dominance is concentrated across different instructions for the integer and floating point codes. Conditional move instructions account for 46.2% of the total dominance cases for the integer benchmarks, while 64% of the dataflow dominance cases lie in multiply instructions for the floating point codes. From these trends, we conclude that dominant operands, as well as the instructions that they render unnecessary, should be accurately predictable with a relatively simple prediction mechanism. In this preliminary work, we only profile the prevalence of dataflow dominance in the Alpha instruction set. Although the type and number of potentially dataflow dominant instructions will vary across instruction sets, we believe that similar dataflow dominance benefits can be attained for other ISAs.

	Spec_INT		Spec_FP			Spec_INT		Spec_FP	
Instruction	**% Dom.**	**% Total**	**% Dom.**	**% Total**	**Instruction**	**% Dom.**	**% Total**	**% Dom.**	**% Total**
AND	14.9%	17.6%	3.5%	0.4%	**CMOVLE**	4.1%	0.0%	67.4%	0.0%
BIC	45.9%	2.2%	57.9%	0.8%	**CMOVGT**	99.3%	0.0%	60.9%	0.1%
ORNOT	0.0%	0.0%	0.7%	0.4%	**CMOVLBS**	33.2%	0.2%	100.0%	0.0%
ORNOT	24.4%	4.4%	4.7%	0.1%	**CMOVLBC**	67.3%	0.1%	9.1%	0.0%
ZAP	1.0%	0.0%	23.3%	0.0%	**MULS**	41.1%	2.8%	21.2%	6.3%
ZAPNOT	46.7%	0.0%	67.1%	0.0%	**DIVS**	0.0%	0.0%	0.0%	0.0%
SRL	0.1%	0.0%	1.1%	0.0%	**MULT**	26.5%	9.7%	16.4%	64.0%
SLL	4.0%	0.0%	0.5%	0.0%	**DIVT**	4.8%	0.2%	2.8%	0.3%
SRA	0.0%	0.0%	0.0%	0.0%	**CPYS**	27.7%	4.6%	41.7%	10.5%
MULL	16.2%	8.2%	21.7%	4.1%	**CPYSN**	14.9%	0.5%	12.9%	0.7%
MULQ	13.2%	1.1%	3.5%	0.2%	**FCMOVEQ**	21.2%	0.4%	58.6%	0.7%‘
UMULH	15.6%	0.4%	17.7%	0.0%	**FCMOVNE**	69.3%	1.2%	32.7%	0.9%
CMOVEQ	62.0%	18.2%	58.7%	6.6%	**FCMOVLT**	0.0%	0.0%	99.8%	0.0%
CMOVNE	72.1%	12.8%	97.2%	1.6%	**FCMOVGE**	0.0%	0.0%	33.5%	0.0%
CMOVLT	96.2%	3.1%	78.3%	2.2%	**FCMOVLE**	0.0%	0.0%	0.0%	0.0%
CMOVGE	40.5%	11.8%	0.8%	0.0%	**FCMOVGT**	0.0%	0.0%	0.0%	0.0%

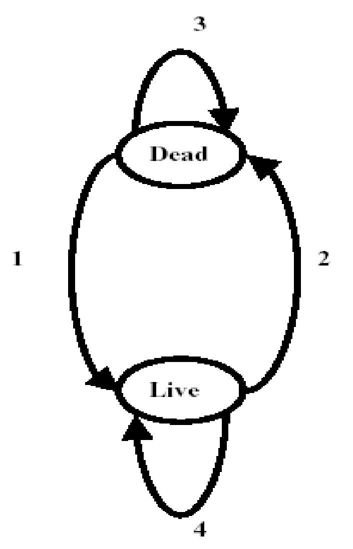
Table 3: Dominant Instruction Characterization. Shown here is the percentage of time that each instruction receives a dominant operand followed by that instruction’s contribution to the total dataflow dominant cases encountered throughout the entire execution.

4 .0 Performance Potential

4.1 Trace-Based Limit Study

In order to evaluate the potential benefits of dataflow dominance exploitation, we performed a trace-based limit study. From each benchmark, we constructed a 50 million instruction trace of the committed instruction stream of the form: "inst, pc, rd#, rs#, rs_dom, rt#, rt_dom, mem_addr." The rs_dom and rt_dom fields hold boolean values that are "1" if the data value in that field is the dominant operand of a dataflow dominant instruction. The traces were collected using the sim-safe simulator from the SimpleScalar Toolset [2] fed by the reference input set that comes with the spec2000 benchmark suite [15]. Each trace is taken from steady-state execution after fast-forwarding 2 billion instructions to avoid initialization effects. A simple pass through each trace provided the data shown in Table 3, which was discussed in the previous section.

As stated earlier, most of the benefits of dataflow dominance come from eliminating the execution of instructions that are used solely to compute a dominated operand. An instruction is proven unnecessary when its output register is overwritten before it is used as a non-dominated operand. In order to prove an instruction unnecessary, the redefinition of its output register must be visible. By applying an appropriate algorithm (discussed in Figure 2) to



Algorithm:

1. All registers start live
2. Apply state machine in reverse program order
3. Examined instruction unnecessary if destination register already dead
4. Restart algorithm on system call

State Transitions:

1. Dead to Live:
- Dead Elimination: Register read
- Transitively Dead Elimination: Register read by necessary instruction
- Dataflow Dominance: Register read by necessary instruction and not dominated

2. Live to Dead:
- Dead Elimination: Register written
- Transitively Dead Elimination: Register read by necessary instruction

3. Dead to Dead:
- Register written

4. Live to Live:
- Register read

Figure 2. Algorithm For Unnecessary Instruction Detection.

the committed instruction traces in reverse program order, we were capable of counting all of the instructions made unnecessary by dataflow dominance. Processing the trace in reverse program order gave us scope bounded only by the length of the trace to locate the redefinition of the instruction being examined.

The percentage of unnecessary instructions that can be eliminated by this algorithm are shown in Figure 3. Here the percentage of unnecessary instructions are graphed for each benchmark when 1) only dead instructions are removed, 2) when dead and transitively dead instructions are removed and 3) when dead and transitively dead instructions are removed along with all instructions made unnecessary by the arrival of a dataflow dominant operand. It is important to remember that the numbers shown here assume a near-optimal detection and verification method for

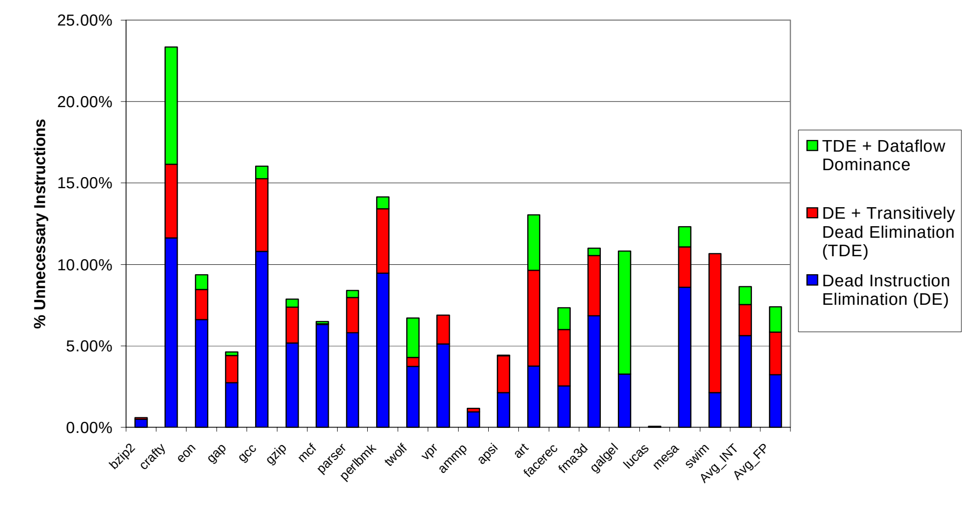


Figure 3. Unnecessary Instructions. Shown here are the instructions made unnecessary by ideal dead instruction elimination, transitively dead instruction elimination, and dataflow dominance exploitation. Our proposed mechanism (see Section 5) targets all three and could achieve these results given ideal prediction and verification.

removing unnecessary instructions. On average, dead instruction elimination removes 4.4% of instructions. An additional 2.3% (as high as 8.5% for swim) of instructions can be removed by targeting transitively dead instructions in addition to dead instructions. When dataflow dominance exploitation is added, an additional 7.5% of instructions can be removed without affecting correct program execution in the best case (1.4% average). If a mechanism were capable of exploiting the benefits of all three removal schemes, necessary instructions could be reduced by as much as 23.3%, or 8.0% on average.

The number of additional instructions removed by exploiting dataflow dominance varies drastically across our benchmarks, from 0% to 7.5%. This is primarily due to four factors: 1) the number of potentially dataflow dominant instructions generated in the compiled code, 2) the percentage of those instructions that contain a dataflow dominant operand, 3) the number of situations where the output of an instruction that is potentially made unnecessary by the arrival of a dataflow dominant operand is also used as an input to another necessary instruction (the instruction therefore cannot be removed), and 4) the fact that in many cases the output of a dataflow dominant instruction is used exclusively as the input to a dead instruction.

The first two cases match intuition; the more dataflow dominance cases, the more benefit that can be gained from their exploitation. The last two cases can lead to some surprising results where a benchmark with a high frequency of dataflow dominant instructions is only able to gain minimal benefit from their presence. The third case reiterates the fact that an instruction deemed unnecessary by the presence of a dataflow dominant operand must be proven to be *exclusively* used for the computation of the dominated operand before it can be removed. If its result is used later in the program, it must be executed along with all instructions that are used to compute its inputs. Many instruction removal opportunities are lost in this manner. The final case describes situations where the dataflow dominant instruction, along with the instructions that would have been unnecessary because of it, are counted as transitively dead instructions because their outputs are used exclusively by dead instructions.

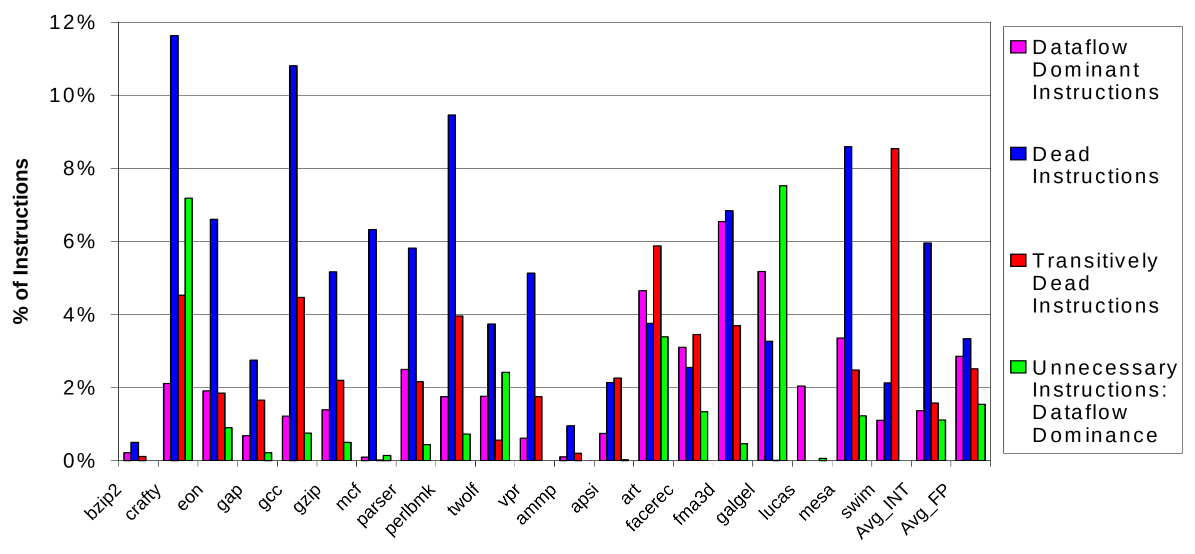


Figure 4. Dataflow Dominant vs. Eliminated Instructions. Shown here is the number of dataflow dominant instructions compared against the number of dead instructions, transitively dead instructions and unnecessary instructions due to dataflow dominance.

These situations allow for the benefits of dataflow dominance to be underestimated when used in conjunction with dead and transitively dead instruction elimination (see Figure 4). Presented here is the same instruction removal data that is shown additively in Figure 3, this time graphed individually against the number of dataflow dominant operands that occur. The number of dataflow dominant instructions is 2.6% on average (as high as 4.7% for art). It is apparent that in most cases each dataflow dominant instruction can result in the removal of less than one additional unnecessary instruction. This is somewhat surprising, but is logical when cases 3 and 4 are considered from above. Though the simultaneous targeting of dataflow dominant and transitively dead instructions can lead to an underestimation of the benefits of dataflow dominance, the combined benefit of their concurrent exploitation exceeds what either optimization could achieve in isolation.

To our knowledge, this work is the first to exploit transitively dead and dataflow dominant instructions, leading to a new contribution of the sum of those two bars in Figure 4. The mechanisms that we propose in Section 5 target all three cases of unnecessary instructions, resulting in the instruction removal totals shown in Figure 3.

4.2 Timing-Based Limit Study

In order to obtain an upper bound on the performance improvement that could be obtained from a realistic dataflow dominance exploitation, we integrated a timing-based simulator, sim-outorder from the SimpleScalar Toolset [2], into our trace-based limit study. In these simulations, we remove all of the instructions identified by our trace study as unnecessary so that they consume no processor resources. The speedup gained from optimal dataflow dominance exploitation is shown relative to that achieved from dead instruction elimination in Figure 5. The simu-

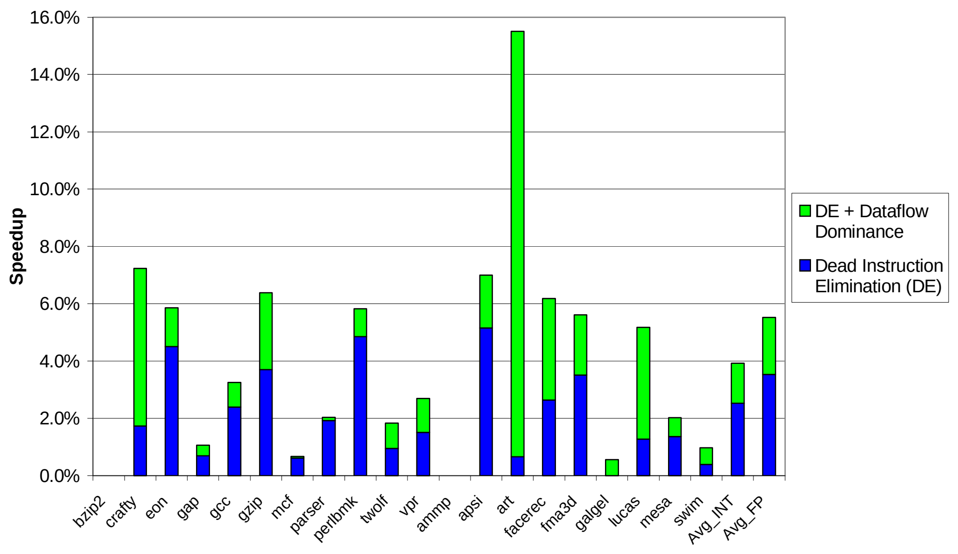


Figure 5. Potential Performance Improvement. Shown here is the speedup gained for our benchmarks using dead instruction elimination and dataflow dominance exploitation, which includes dynamically dead instructions elimination. Also shown are the processor simulator parameters used in all of our experiments.

lated processor configuration for all of our experiments is described in Table 4. For our benchmarks, dead instruction elimination results in an average speedup of 5.1%. Speedups gained by then adding optimal dataflow dominance exploitation are non-trivial, as high as 14.9% for art. Our proposed mechanisms, (Section 5) designed to take advantage of dataflow dominant instructions, should easily be able to remove dead instructions as well. With such a mechanism, speedups as high as 15.5%, 7.3% on average, over normal execution can be attained.

The performance improvement gained by the exploitation of dataflow dominance comes primarily from two sources: 1) reduced resource contention within the processor that results in fewer structural hazards and 2) removal of

- **Front End:** 4 wide fetch, 16KB bimodal 2-level branch predictor, 16 entry return address stack
- **Instruction Window:** 16 entry with 64 entry ROB, 4-wide issue, 64 rename registers, 4-wide retirement
- **Execution:** 4 int ALU (2 mult.), 2 fp ALU (2 mult.)
- **Memory Hierarchy:** 8 KB DL1 (2-way, 16B blocks, 2 cyc. latency), 32KB IL1 (2-way, 32B blocks, 2 cyc. latency), 512KB unified L2 (4-way, 64B blocks, 15 cyc. latency), 500 cycle memory latency

Table 4: Simulated Processor Parameters

memory system effects from unnecessary loads. When unnecessary instructions are removed, they do not consume resources critical to the speed of execution, such as reorder buffer and load-store queue slots, functional units, and fetch, decode, and commit bandwidth. These freed resources allow instructions on necessary paths to execute more quickly. The removal of unnecessary loads leads to improved memory performance both by removing the cache misses caused by unnecessary instructions and by increasing the effective cache size available to necessary loaded data blocks. Figure 6 shows the reduction of level-1 data cache misses that results from dead instruction elimination

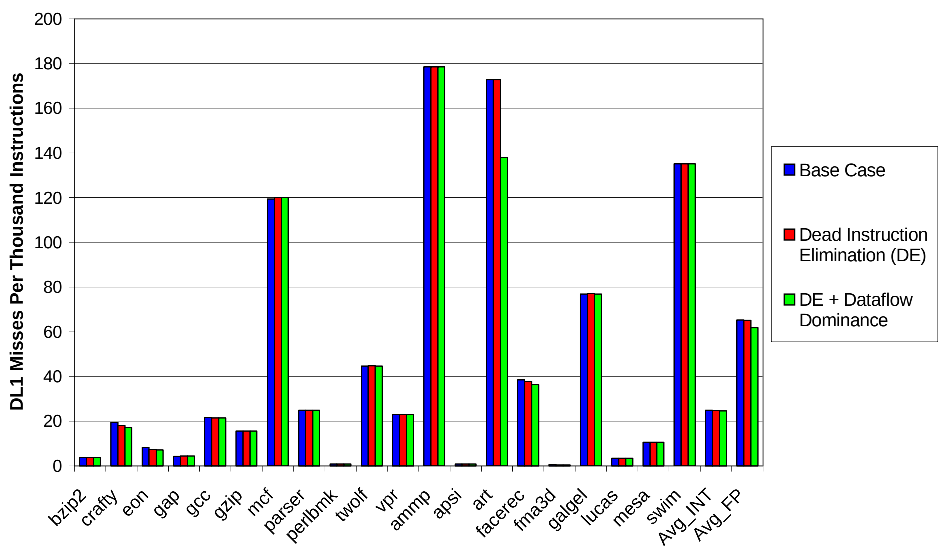


Figure 6. L1 Cache Behavior. Shown here are the number of L1D cache misses per thousand instructions for dead instruction elimination and dataflow dominance exploitation.

and dataflow dominance exploitation. This data shows that the benefits of dataflow dominance do not consistently come from the same source. Several benchmarks show little or no reduction in cache misses but still receive a performance boost from dataflow dominance (see Figure 5). These benchmarks rely on the reduced processor resource contention gained by unnecessary instruction removal to gain performance. Removing unnecessary loads leads to significant cache miss savings in art, leading to its significant speedup.

4.3 Instruction Removal Issues

Not surprisingly, the complete removal of instructions from a program is quite difficult and is accompanied with a set of consequences. In this section, we discuss these issues and propose solutions to overcome them.

4.3.1 Unnecessary Instruction Detection and Verification

As illustrated in Figure 2, in order to prove that an instruction is unnecessary it must be verified that the instruction's output register is redefined before it is read as an input to a necessary instruction. In order for this to be verified, all of the instructions up to that register kill must be kept in the machine for potential examination. If the distance between the instruction being examined and the redefinition of its output register is excessively large, no realistic hardware mechanism could detect the register kill. In those cases, truly unnecessary instructions would not be identified. Figure 7 shows the static instruction distance between instructions deemed unnecessary by a dataflow dominant instruction and the redefinition of their output registers. Only the integer benchmarks are shown for brevity, but similar instruction distances hold for all of our benchmarks. On average, a scope of 256 instructions captures 70% of the unnecessary instructions and a much higher percentage for most benchmarks. This data shows that the majority of the benefit from dataflow dominance can be achieved in a reasonable sized machine (reorder buffer size, LSQ size, etc.). Similar static instruction distance trends hold for dead and transitively dead instruction elimination.

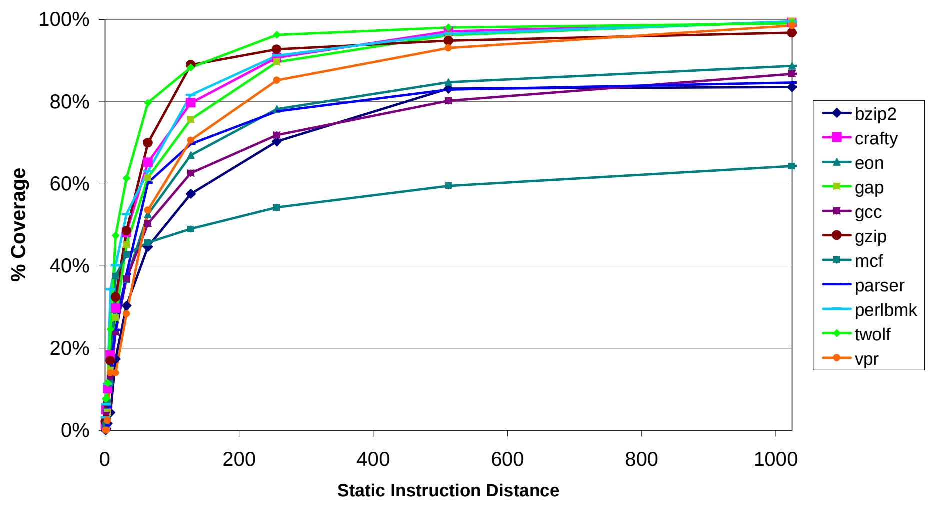


Figure 7. Detection Distance. Shown here are the cumulative distributions of the static distance between the definition of a register and its redefinition (kill) that needs to be visible in order for the instruction to be proven unnecessary by dataflow dominance.

4.3.2 Exception Behavior

Dynamically eliminating instructions from the execution of a program also leads to the elimination of these instructions' potential to cause exceptions. Situations could arise when an instruction would have caused an exception during normal execution but when the dataflow dominance optimization is turned on the exception never occurs because the instruction is never executed. For user code, this is only a problem if an exception handler has been attached to the program, hence providing a method by which the user could use the exception as a condition to enter another piece of code. Therefore, a processor equipped with a mechanism to take advantage of any of the previously discussed instruction removal techniques should disable these optimizations when a software exception handler is attached to the program. A runtime system that is aware of both this technique and of currently active exception handlers could accomplish this goal.

5 .0 Implementation Issues

Up to this point, we have defined dataflow dominance abstractly without regard to specific hardware mechanisms. This section discusses several design alternatives in which a traditional processor could be augmented to exploit dataflow dominance as well as the implementation issues surrounding such techniques.

5.1 Detection

The first step in removing unnecessary computations is to identify candidate instructions that are likely to be useless. As discussed in Section 3, in order to guarantee that an instruction is unnecessary and can safely be dropped from execution, it must be ensured that no necessary instruction consumes the value it produces. This implies that the subsequent definition of the candidate instruction’s destination register must be observed in order to know that the value is dead. We propose a FIFO queue structure that instructions are pushed onto as they retire from execution in the machine. This allows a history of committed instructions to be built up on which to perform the dataflow analysis described in Figure 2. Figure 7 provides some insight into how queue size affects the detection coverage. It indicates that a 256-entry queue is sufficiently large enough to capture greater than 70% of all unnecessary instructions.

5.2 Exploitation and Verification

The previous sub-section described a mechanism that can determine if an instruction was necessary given the dynamic, committed instruction stream that it appeared in. However, this postmortem analysis only allows us to determine if it was necessary in hindsight -- this dynamic instruction cannot be eliminated because it has already finished execution. Instead, we can leverage this information by remembering our observation the next time the instruction sequence is observed. We propose two hardware mechanisms to exploit this knowledge: one based on a traditional predictor structure, and one based on existing trace cache proposals [9][10][13]. We provide a qualitative

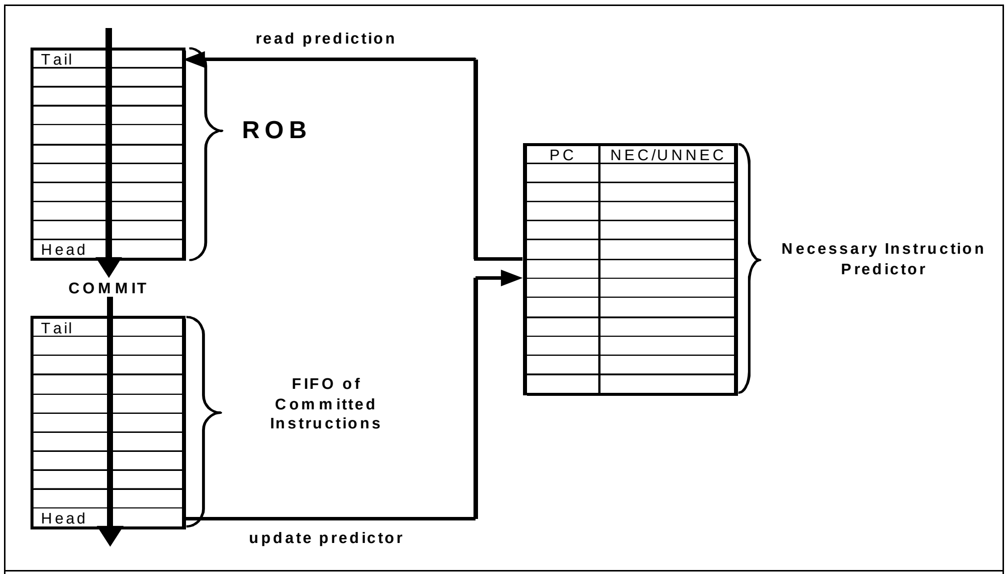


Figure 8. Predictor Based Implementation Diagram

discussion for completeness; a more rigorous evaluation of these techniques is left to future work.

5.2.1 Predictor Based Instruction Removal

In order to make room for recently-committed instructions in the FIFO, entries are popped from the head and analyzed in order to determine if it was functionally necessary for them to execute. If an instruction was determined to be unnecessary, a predictor structure is updated to indicate that we should attempt to eliminate this instruction on subsequent encounters. A diagram of our proposed predictor based instruction removal mechanism is shown in Figure 8. Initial studies (the results of which are not presented here) indicate that a simple two-bit predictor indexed by program counter can achieve greater than 98% accuracy. Alternately, the large amount of prior work regarding path-based branch prediction exists and can be leveraged to achieve a higher degree of accuracy [16].

As instructions are fetched, the predictor is indexed to predict whether or not they are necessary. If they are predicted necessary, execution continues as it would normally. If they are predicted unnecessary, then their execution is deferred (by assigning a low scheduling priority) in order to enable a greater scope of younger instructions to enter the machine. This allows a history of subsequent instructions to be built and increases the likelihood of observing the unnecessary conditions discussed in Section 4.1.When the candidate unnecessary instruction is eventually selected for execution by the scheduler, the younger, in-flight instructions are examined to determine if the candidate can be removed. If the unnecessary conditions are met, the instruction is dropped from execution. Otherwise it is allocated a functional unit and continues down the pipeline as it would be normally.

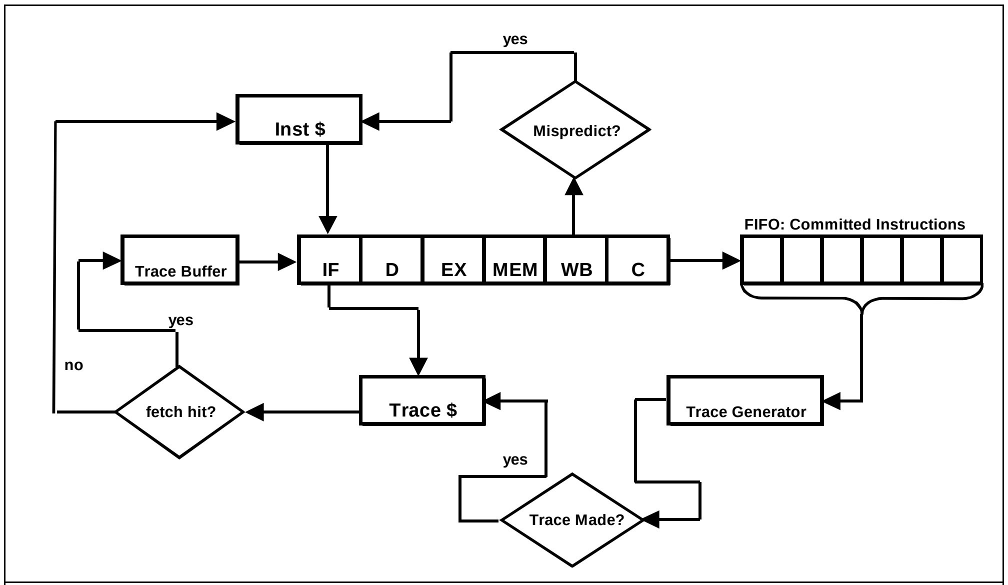


Figure 9. Trace Cache Based Implementation Diagram.

5.2.2 Trace Cache Based Instruction Removal

While the predictor-based mechanism proposed in Figure 5.2.1 provides a simple method to remove unnecessary instructions from execution, it does not achieve maximal benefit compared to other schemes because the eliminated instructions still consume some degree of execution resources. Even if an instruction is eventually squashed from execution, it still consumes memory and fetch bandwidth, and is still allocated a reorder buffer (ROB) entry, rename register, issue queue slot and potential load/store queue slot, all of which can be consumed while the determination of the instruction's necessity is pending.

The intent of the mechanism proposed in this section (shown in Figure 9) is to avoid such resource consumption by unnecessary instructions. We leverage structures and algorithms first proposed by instruction trace caches [9][10][13]. Committed instructions are pushed onto a FIFO as they were in the prediction-based mechanism. However, instead of determining if instructions are necessary in isolation as they are popped off of the queue, we collectively decide whether each instruction is necessary only by examining younger instructions in the queue. This means that while in the prediction-based approach each instruction has a scope of *q* younger instructions to determine if it is necessary (where *q* is the number of entries in the FIFO), in this approach only the head entry has visibility to *q* instructions, and younger entries have visibility of *q - i*, where *i* is the distance to the head of the queue.

Once a sufficient number of unnecessary instructions have been observed in the queue, a *trace* of up *q* dynamic instructions is generated and the unnecessary instructions are removed. The trace is inserted into a *trace* *cache* that is indexed by the starting fetch address of the dynamic instruction stream. Subsequent fetch requests are searched for in the trace cache before (or in parallel with) the instruction cache. If the fetch address hits in the trace cache, all instructions in the trace are inserted into the front end of the machine. The unnecessary instructions do not exist in the trace and therefore consume no execution resources, unlike the previous technique. If no trace entry is found, instructions are fetched from the I-cache or memory.

Finally, it must be verified that the removed instructions are not, in fact, necessary. This involves guaranteeing that the control flow follows the same execution and that any instructions that were dataflow dominant when the trace was constructed are still dataflow dominant when the trace is replayed. Thus branch outcomes and dataflow dominant conditions are embedded into the trace when it is constructed. If an inconsistency is detected while replaying instructions from the trace, the entire trace is discarded and fetching resumes from the I-cache normally. This notion of dynamic optimizations of an atomic instruction stream is similar to that proposed in the rePLay framework [11].

6 .0 Conclusion

This work introduces the notion that many dynamic instructions can compute their result without knowledge of all of their source operand values. We define this characteristic as *dataflow dominance*, and show that the instructions that compute the unnecessary source operands are not required to execute and can safely be removed. We highlight the conditions that must be satisfied in order to remove such instructions, and show that up to 23.3% of dynamic instructions can be removed, leading up to an idealized speedup of 15.5% in SPEC2000 benchmarks. We further characterize some of the fundamental behavior of programs that exhibit a large portion of dataflow dominant operations and discuss how such characteristics affect their detection and exploitation. Finally, this work propose two hardware mechanisms to remove instructions that are unnecessary due to dataflow dominance, and we discusses several tradeoffs among these designs.

7 .0 References

[1] Gordon Bell, Kevin Lepak, and Mikko H. Lipasti. A characterization of silent stores. In *Proceedings of PACT-2000*, Philadelphia, PA, October 2000.

[2] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin Computer Sciences, 1997.

[3] J. Adam Butts and G. Sohi. Characterizing and predicting value degree of use. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, October 2002.

[4] J. Adam Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and operating Systems (ASPLOS-X)*, October 2002.

[5] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[6] Kevin Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of ISCA-2000*, Vancouver, June 2000.

[7] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (AS-PLOS-VII)*, October 1996.

[8] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, December 1997.

[9] Sanjay J. Patel. Trace cache design for wide-issue superscalar processors. Thesis, University of Michigan, 1999.

[10] Sanjay J. Patel, Daniel h. Friendly, and Yale N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical report, University of Michigan, May 1997.

[11] Sanjay J. Patel and Steven S. Lumetta. replay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[12] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical report, North Carolina State University, November 1999.

[13] Eric Rotenberg, Steve Bennett, and Jim Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, December 1996.

[14] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Maynard, MA, 1992.

[15] Systems Performance Evaluation Cooperative. SPEC benchmarks. http://www.spec.org.

[16] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM Press, 1991.

[17] Craig Zilles. Master/slave speculative parallelization and approximate code. Thesis, University of Wisconsin, 2002.