DISE: Implementing Application Meta-Features via Software-Programmable Decoding*

Marc L. Corliss   E Christopher Lewis   Amir Roth
Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104 USA

November 18, 2002

Abstract

Dynamic Instruction Stream Editing (DISE) is a cooperative software-hardware scheme for efficiently adding *meta-features*—*e.g.*, safety/security checking, profiling, and dynamic code decompression—to an application. DISE implements meta-features by providing a programmable processor decoder that macro-expands certain instructions into parameterized instruction sequences. The technique is similar in spirit to programmable microcode and exploits the macro-expansion-style decoding technology already present in many of today's processors.

DISE is a single facility that unifies the implementation of a large class of meta-features previously requiring either special-purpose hardware widgets or software-only tools like binary rewriters. Meta-features implemented via DISE require no new hardware widgets and avoid many of the costs of binary rewriting, primarily static code bloat and the fixed overhead of the rewriting process itself. The dynamic character of DISE makes it appropriate for meta-features with fine grain intermittent usage patterns (*e.g.*, sampled profiling) as well as applications that have no natural software implementations (*e.g.*, dynamic code decompression).

In this paper, we introduce DISE and show how it can be used to formulate three meta-features: path profiling, memory fault isolation, and dynamic code decompression. We find DISE to be general, powerful, and efficient. We describe the requirements of a DISE implementation and show one particular design. We also discuss the implications of DISE on all facets of a system. Using cycle-level simulation we show that DISE-implemented meta-features have competitive raw performance and a profitable dynamic cost structure.

1 Introduction

Today's systems employ a wide array of techniques that can broadly be termed *application meta-features*. Meta-features are added to a *principal application* to enhance its value in some way, *e.g.*, ensure its safety, improve its reliability, or reduce its execution latency. Meta-features are used for diverse purposes and in many computing environments. Examples include safety checking (mobile code), profiling to identify optimization opportunities (high-performance processors), and decompression (cost-conscious embedded processors).

Due to their high utility in many domains, considerable research effort has been devoted to implementing meta-features in both hardware and software. Hardware approaches typically implement meta-features using dedicated (potentially programmable) pipeline stages, while software approaches embed meta-features into the principal application itself in the form of additional instructions. The hardware and software approaches have complementary strengths and weaknesses. Hardware implementations exact no overhead on the principal application, but are functionally rigid. Even programmable designs are restricted by the number of stages allocated to the function, their position in the pipeline, and the kinds of basic operations each can perform. Software implementations are functionally rich but have a fixed, high cost structure. The additional instructions bloat both the static instruction working set and the dynamic instruction count, reducing the effective instruction cache size and pipeline throughput, respectively. Furthermore, the

*This is a revised version of University of Pennsylvania Department of Computer and Information Science Technical Report MS-CIS-02-24, June 2002.

rewriting process itself exacts a fixed cost that constrains the granularity at which binary rewriting can be effectively used. Certain meta-features, *e.g.*, dynamic code decompression, have no natural efficient software implementations.

We propose a new cooperative software-hardware approach to implementing meta-features, one that has the rich functionality of software approaches, without the high and fixed costs of inserting those instructions into the static executable. Our approach is to formulate meta-features as *dynamic instruction stream transformations*. Meta-features are programmed by specifying a set of rules or "macros" for replacing instructions that obey certain criteria with parameterized instruction sequences. The processor's decoder executes the specification on the principal application, feeding the execution engine an instruction stream that includes meta-feature code. Inserting meta-feature code at the decode stage is key. Pre-execute code insertion enables meta-features that modify the principal application, not just observe. Post-fetch code insertion sidesteps the costs that result from inserting meta-feature code into the principal application's static binary first.

Decoder-based macro-expansion is used in virtually all IA-32 compatible processors to dynamically convert each IA-32 CISC instruction to one or more internal RISC primitives [15, 10, 14, 13]. We co-opt this technology for meta-feature implementation by making the macro-expansion rules programmable. We call this form of programmable decoding *DISE (Dynamic Instruction Stream Editing)*. Like hardware meta-feature implementations, DISE allows meta-features to evolve as the program runs, and has no cost associated with transforming the principal application binary. Like software-only approaches, DISE is general and enables flexible processor resource allocation—meta-features and principal applications share the same execution resources, allowing the latter to use all resources when meta-features are not employed. In this paper, we introduce DISE, describe a sample implementation, and discuss the implications of programmable decoding on meta-feature portability and system security.

DISE is a single facility that unifies the implementation of a large class of meta-features. Although instruction-level macro-expansion is best suited for implementing local (*i.e.*, peephole) transformations, many interesting meta-features—including all the ones listed above—can be formulated as such. We show DISE formulations for path profiling, memory fault isolation, and dynamic code decompression. Cycle-level simulation of the SPEC2000 integer benchmarks shows that the DISE-implemented meta-features have advantages over both hardware and software counterparts. DISE memory fault isolation and path profiling exact less performance overhead on the principal application than the corresponding software-only implementations (*e.g.*, binary rewriting), and this is without accounting for the initial cost of the transformation process itself. DISE's ability to toggle meta-feature functionality at fine granularity is also useful for applications, like sampled path-profiling, which have intermittent usage patterns in dynamic settings. DISE-based code compression/decompression results in significant compression factors and improved performance.

The remainder of this paper is organized as follows. The next section summarizes related work. Section 3 presents and discusses DISE and a sample compartmentalized implementation requiring only decoder changes. Section 4 demonstrates the utility of DISE via three applications: software-based fault isolation, path profiling, and decompression. Section 5 presents a performance evaluation. The final section contains conclusions and proposals for future work.

2 Related Work

Decoder-based translation. Most recent IA-32 compatible processors [14, 13, 10] use the decoder to dynamically macro-expand each IA-32 CISC instruction into one or more internal RISC operations, with the Pentium4 [13] caching expanded instructions. Dynamic Instruction Formatting (DIF) [23] proposes not only to cache translated instructions, but schedule them into VLIW groups first. Speculative Decoding (SD) [18] is a recently proposed technique that effectively implements execution time optimizations like silent-store elimination and load merging using alternate macro-expansions. From a mechanical standpoint, DISE is nearly identical to these decoders/translators. However, the latter are completely inaccessible to software, and capable only of changing/optimizing internal instruction representations, not adding functionality. DISE adds a software-programmable interface to the basic decoding/expansion facility to enable the implementation of general application meta-features.

Hardware implementations of meta-features. Early hardware implementations of meta-features exploited programmable microcode [8, 24]. Initially used to augment the ISA with application-specific control for improving datapath utilization, programmable microcode was later co-opted for implementing meta-functionality like address tracing [1]. Although microcode remains a viable option for implementing complex instructions and programmable microcode stores persist (*e.g.*, Intel's P6 supports a limited mechanism for patching microcode to fix bugs in the field [16]), its current use in processors is too sparse and irregular to effectively support meta-functionality. DISE

implements meta-functionality using conventional instructions which execute on existing datapaths under existing hardwired control, enabling a richer set of meta-features.

Recent proposals implement meta-features using dedicated, potentially programmable, pipeline stages. Examples include the profiling processor [34], the instruction path co-processor (ICOP) [9], which has been applied to both profiling and trace construction/optimization, and the DIVA checker [2] which provides detection of transient and design errors. These provide meta-functionality at virtually no cost to the principal application, but are quite inflexible. In particular, the dedicated stages are placed post retirement to minimize their performance impact and thus are inappropriate for meta-features which must inspect and potentially modify program instructions *before* they execute. By transforming the instruction stream before execution, DISE supports a different (potentially broader) class of meta-features.

In decoder-based decompression [21], a decoder interprets tagged instructions in a compressed principal application as decompression-dictionary indices and replaces them by the corresponding entries. DISE generalizes this functionality by allowing parameterized interpretation (matching) and replacement of instructions, and thus can be used for more than just compression.

Since DISE deals with meta-features, it has little relationship to application-specific hardware including reconfigurable hardware. DISE reconfigures the instruction stream, not the hardware.

Instruction translation in software. Instruction translation and macro-expansion has also been performed in software. Systems like FX!32 [31], DAISY [12] and Transmeta's Crusoe design [19] use software to convert one ISA to another, potentially caching the translation. Systems like Dynamo [3] add optimization to the translation loop. Meta-features may be added to a software translator but adding them dynamically to translation code may be difficult, and the software translation process may be too heavy or coarse-grained for some meta-features. At the same time, DISE is probably not well suited for wholesale ISA translation, but rather to meta-features that can be programmed as simple rules that apply to a few instruction classes.

Software implementations of meta-features. Binary rewriting tools that do not translate the ISA but provide simple hooks for adding meta-features to an application include Atom (Alpha) [30], Etch (IA-32) [27], and EEL (SPARC) [20]. They have been successfully used to implement profiling [5], dynamic race detection [28], and software simulation of hardware features like shared memory [29] and memory fault isolation [32]. Paradyn includes a binary rewriting system that can transform any program, including the operating system kernel, while it is running [22]. These meta-features can all be implemented using DISE. The DISE implementation may require more dynamic instructions (*e.g.*, the software version may exploit static analysis to optimize meta-feature code), but would not degrade instruction cache performance, would not incur the fixed overhead of rewriting, and would enable fine granularity usage patterns.

3 DISE: Dynamic Instruction Stream Editing

In this section, we present the DISE mechanism, beginning with its core functionality. We describe the matching and replacement processes, show how they can be implemented within the structure of a conventional decoder, and argue for the inclusion of DISE dedicated register storage. Next, we address the semantics of DISE replacement instructions including precise state and the mechanics of control transfers around, out of, and within replacement sequences. We close with a discussion of DISE as a system utility. We discuss the DISE programming interface, the visibility of DISE structures to the operating system and the resulting portability issues, and the implications of programmable decoding on security.

3.1 Core DISE Functionality: Matching and Replacement

DISE inspects every principal application instruction (*principal instruction* for short) and macro-expands those that match specified criteria. We call expanded principal instructions *triggers*, and the macros themselves *replacement sequences*. This functionality is provided via three logical components: the *pattern table (*PT*)*, the *replacement table (*RT*)*, and *instantiation logic*.

Basic structures. The *pattern table* (PT) is a small, fully-associative table that contains trigger specifications against which all principal instructions are matched. Each PT entry contains four fields: 1) a *pattern*, 2) an *enabled* bit, 3) a *replacement sequence index*, and 4) a *replacement sequence length*. A pattern specification may include any combination of opcode, opcode class (*e.g.*, "all stores"), input or output logical register names, immediate field or any attribute thereof (*e.g.*, "a negative immediate value"). DISE is able to specify triggers of the form "loads that use

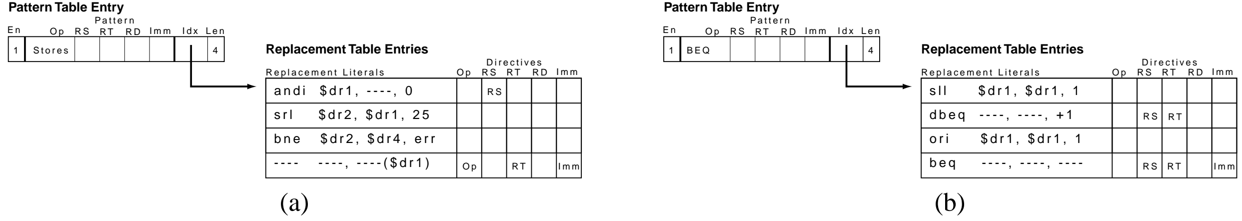


Figure 1: Sample DISE productions from (a) memory fault isolation and (b) path profiling.

the stack-pointer as their base address" or "conditional branches with negative offsets." Future meta-feature DISE implementations may require patterns that specify non-instruction attributes like the PC or the predicted direction of a branch. We have not found uses for this functionality so far.

The *replacement table (*RT*)* is a RAM which houses the replacement sequences. To enable interesting application meta-features, replacement sequences are *parameterized*. Each RT entry contains a *replacement literal*, and a series of *instantiation directives* one for each field of the instruction. The replacement literal is a bit string representing one replacement instruction in internal decoded format (this does not mean that the internal format of decoded instructions is visible to software as we discuss in Section 3.3). Non-empty instantiation directives instantiate fields in this instruction with fields from the trigger. Instantiation directives are interpreted differently for different fields. For instance, register fields have four valid directives: a literal (empty) directive, and three directives for selecting any of the three registers named in the trigger instruction. With parameterization, (contrived) transformations like the following are possible: "replace all stores with loads using the same base register and offset" or "swap the two input registers of all divide instructions." Although non-instruction attributes are not useful in matching, we have found uses for them in replacement. Specifically, the ability to encode the trigger's PC as a replacement instruction immediate has applications which we discuss in Section 4.2.

The *instantiation logic* is a combinational circuit that executes instantiation directives on replacement instructions using fields extracted from the principal trigger.

DISE dedicated registers. DISE allows replacement instructions to access *dedicated registers* that are not visible to the ISA. The ability to use an expanded register set is quite useful. It allows replacement instructions to use temporary registers without saving and restoring user registers. More importantly, it provides persistent register storage across individual expansions allowing global meta-features to be synthesized from peephole, single-instruction expansions, without requiring the compiler to reserve user registers. A dedicated register space greatly simplifies meta-feature formulation and reduces the cost of DISE meta-feature implementations.

An example production. Two sample DISE productions are shown in Figure 1. For illustration purposes, we use the MIPS ISA. The first production (a), taken from a memory fault isolation meta-feature (Section 4), augments every store with instructions to check the validity of the address. A single pattern matches all stores and replaces them with four instruction sequences. The first copies the store base address into a dedicated register `$dr1`. This instruction has a literal opcode and immediate, but is parameterized to replace its input register RS with the address register RS of the trigger store. The second and third instructions are entirely literal, shifting the address and comparing the result to another dedicated register (`$dr4`). The final instruction is the principal store, except that the address register is "renamed" to `$dr1`; here we have a literal address register `$dr1`, but copy the opcode, data register, and address offset from the original store. The second sample production (b) is used for path profiling (Section 4). It specifies replacements and literals similarly, but the second replacement instruction specifies an intra-replacement sequence branch (Section 3.2).

Sample implementation. DISE can be added in an nonintrusive way to a processor's decoder, whether it be RISC or CISC, single-cycle or pipelined. Since DISE macro-expansion is mechanically similar to the instruction-to-internal-micro-instruction translation performed by most IA-32 processors [14, 13, 15, 10], we explain how DISE is added to an IA-32 decoder.

IA-32 translation happens in one of two ways. Simple translation—producing four or fewer micro-instructions—is done via combinational logic arrays (CLAs). Translation requiring five or more micro-instructions is performed by reading and instantiating templates from a micro-instructions ROM (μinst ROM). The decision on whether and at what offsets to use the μinst ROM is based on the IA-32 opcode and performed by a second, IA-32 opcode-indexed ROM (the selector). In a single-cycle decoder implementation, the CLAs, μinst ROM, and selector are accessed in

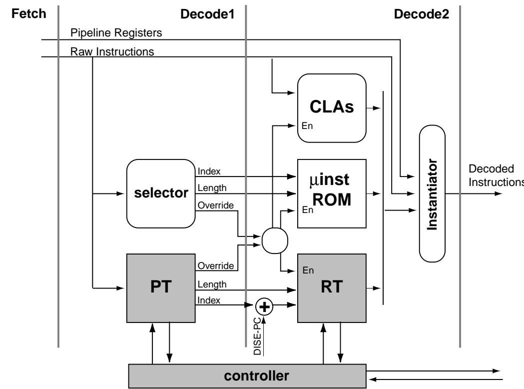


Figure 2: Sample DISE implementation within a two-stage decoder.

parallel. The selector asserts an override when when the μinst ROM must be used rather than the CLAs, resulting in a single-cycle stall. In a pipelined implementation (P6 [14] and K7 [10] have 3-stage decoders), the selector is placed one stage ahead of the CLAs and μinst ROM, and the stall is avoided.

The DISE structures parallel the existing IA-32 structures. The RT parallels the μinst ROM in both placement and structure. The only reason we don't unify the two is that turning the μinst ROM into a RAM is perceived to be too risky. The RT and μinst ROM share the instantiation logic. The PT parallels the selector table, but is a fundamentally more complex structure: it is content associative and matches full instructions, not just opcodes. The PT override signal (which indicates RT expansion) has priority over the selector's μinst ROM override. Figure 2 shows the three DISE structures in a two-cycle IA-32 decoder.

The RISC/CISC issue also arises in the PT matching implementation. On architectures with regular ISA encodings, matching may be performed by masking and comparing raw instruction bits. On architectures with irregular encodings, instructions may need to be partially decoded and reformatted before PT access. It is likely that processors with irregular ISAs already contain such pre-decode facilities [14].

3.2 Precise State, Control, and DISE-PC

The execution engine does not distinguish DISE replacement instructions from their principal counterparts. However, replacement instructions are different when it comes to sequencing. Principal instructions are sequenced by the fetch engine via control speculation, while replacement instructions are sequenced by the decoder using macro-expansion of a single PC. These distinctions come into play when the execution engine must send sequencing feedback to the fetch engine. Two kinds of events require execution to redirect fetch: resolution of mispredicted branches and restartable interrupts (*e.g.*, page faults). When a page fault is triggered by a principal instruction, the pipeline is flushed, the fault is repaired by the OS, and fetch resumes at the faulting PC. But where does fetch resume after the OS repairs a replacement instruction fault? And what does it mean for a replacement branch to be mispredicted when it was never predicted to begin with? These questions point to the issues of precise state and control within replacement sequences. We deal with these here.

DISE-PC. DISE maintains an additional state element called the *DISE program counter (*DISE-PC*)*. A replacement instruction's DISE-PC is its offset from the start of the replacement sequence. For uniformity, pipeline control associates a DISE-PC with *every* instruction, not only replacement instructions. Where previously tagged by PC, instructions are now tagged by PC:DISE-PC pairs. Unexpanded instructions have PC:0 tags. Replacement instructions have PC:DISE-PC tags where PC is the PC of the principal trigger.

Precise state within replacement sequences. Precise state within replacement sequences—and hence precise restarts after replacement instruction faults—is implemented via the DISE-PC. When a replacement instruction at PC:DISE-PC faults, fetch is restarted at PC:DISE-PC. The fetch engine ignores the DISE-PC annotation, fetching the principal instruction at PC. The DISE decoder recognizes the annotation and instantiates the replacement sequence starting at DISE-PC, effectively skipping the first DISE-PC-1 replacement instructions. In fact, for a given matching PT entry, the RT always instantiates replacement instructions starting at PT.OFFSET + DISE-PC and ending at PT.OFFSET + PT.LENGTH.

Principal branches within replacement sequences. Control transfers using principal branches take place at the principal instruction level, use PCs only, and are oblivious to the presence of DISE-PCs (and DISE in general). The target PC of a principal control transfer is any PC in any form: absolute or relative, encoded in the executable or computed at runtime, stored in memory or otherwise. The target DISE-PC is implicitly 0. This arrangement disallows one principal instruction from transferring control into the middle of a replacement sequence of a second principal instruction, but such transfers have no apparent practical uses and break the notion that a replacement sequence is properly contained within a principal instruction.

A subtle issue surrounds principal branches that are embedded within replacement sequences. The question is: do replacement instructions that appear after the principal branch belong to the taken path of the branch or the non-taken path? The answer is a counter-intuitive, "*neither*." These instructions are associated with the branch itself and they are only executed if the branch is *correctly predicted*. If the branch is mispredicted, the subsequent replacement instructions are flushed and not refetched as control is rerouted to the instruction that logically follows the branch (fall-through or target). Although counter-intuitive, these semantics are consistent with DISE control flow. Our implementation of memory fault isolation (Figure 1(a) exploits a variant of this behavior. Here DISE inserts a principal branch to error code prior to the store. Since the branch is not fetched, its default prediction is "not-taken." At runtime, if the branch is taken, the store is flushed and control is transfered to the error routine. This is the desired behavior.

Control within replacement sequences and DISE calls. The DISE-PC enables a powerful model for control within replacement sequences which is completely distinct from principal application control.

DISE provides special instructions—variants of branches and jumps—that modify the DISE-PC only. The DISE decoder implicitly assumes that all DISE control transfers, even unconditional ones, are not-taken. Executing a replacement sequence-internal control transfer is mechanically identical to restarting from a replacement sequence internal fault: the execution engine is flushed and fetch is restarted at PC:new-DISE-PC. DISE control transfers can be used to implement conditionals and even loops within replacement sequences. Our implementation of path profiling (Figure 1(b)) exploits a replacement-sequence internal branch.

DISE also allows replacement sequences to include calls to principal application functions. Principal function code executes unaware that it was called from within a replacement sequence, and on termination returns control to the proper replacement instruction. This feature is quite useful for implementing complex meta-features. Rather than implement a complex replacement sequence, the meta-feature is implemented via principal application code which the replacement sequence calls. *DISE calls* are implemented using a special variant of the call instruction. The DISE dedicated registers are used to save/restore the ISA caller-registers, freeing them to implement the calling convention. DISE itself is disabled under a DISE call. This is often the desired functionality, as instructions within the called function are thought of as part of the replacement sequence, not as candidates for replacement themselves. Failure to disable DISE under a DISE call could result in bottomless macro-expansion recursion. It is the equivalent of rerouting replacement instructions back through the decoder for further expansion. Disabling DISE under a DISE call also preserves the DISE-PC and allows return to the proper replacement instruction. Figure 8(a) in the Appendix illustrates this.

3.3 The DISE System Utility

We close this section by discussing DISE's role as a system level utility. DISE exposes non-ISA machine features (*e.g.*, DISE specific instructions, registers, and microarchitectural pipeline register values) and allows code that accesses these features to be dynamically added to user-level applications. This apparent breach of the ISA abstraction opens the door for both incompatibility problems and security violations. We deal with these issues here.

Security and access to DISE structures. DISE does not endanger processor internal state. Replacement instructions execute via the processor's data paths and cannot create inconsistencies in its control logic. In addition, the only microarchitectural structures accessible to DISE are either DISE specific (*e.g.*, the DISE registers) or can only be read (*e.g.*, pipeline register values).

The more material security risk presented by DISE is the potential for one process, via DISE productions, to examine and possibly interfere with the state of a second process, perhaps even the OS kernel. Our general strategy for dealing with security issues is to restrict direct DISE access to privileged code (*i.e.*, the OS kernel itself). Since most application meta-features have a system utility flavor, this restricted access is appropriate. If user-level access to DISE is needed in the future, the OS could export a DISE API. Within the implementation of this API, the OS could check replacement sequences and reject those which it deems unsafe. As a further security measure, by controlling

the contents of the DISE structures across context-switches, the OS can "sand-box" an application so that it only adds meta-features to its own code.

The OS accesses DISE structures in two ways. The data elements—the DISE registers and DISE-PC—are accessed via existing datapaths. Access is specified by DISE replacement instructions, and can be extended to the OS via privileged ISA versions of the same. For portability reasons, the PT and RT are accessed indirectly via a microcoded controller (more on this shortly). OS access is used to both implement meta-features and to preserve user-defined meta-feature state—should such be implemented—across process switches.

Portability and DISE programming. While the DISE registers contain plain data (*i.e.*, 32 or 64 bit values), the PT and RT contain information that is far more implementation dependent: instruction pattern specifications and replacement instructions in internal decoded form. For portability, access to the PT and RT is provided via a logical interface that is implemented by a microcoded controller. The controller interface allows PT and RT entries to be written, read, disabled, and deleted; and it exports the number of entries available in each structure. The controller interface may be instruction based or memory mapped.

The interface/controller combination allows DISE meta-feature programmers to specify patterns, replacement instructions, and instantiation directives in a logical way, without knowing the internal pattern representations or decoded instruction formats. The controller translates logical productions to physical PT/RT bits and vice versa. This arrangement effectively makes DISE meta-features portable, perhaps even across architectures.

4 DISE Meta-Feature Implementation Examples

DISE enables the implementation of a large class of meta-features. In this section, we present three well-known and well-studied meta-features—memory fault isolation, path profiling, and dynamic code decompression—and describe DISE implementations of each. Although none of these meta-features are new, their implementation via a single hardware facility is. We conclude with a sketch of several other meta-features amenable to DISE implementation.

4.1 Memory Fault Isolation

Memory fault isolation prevents multiple applications from interfering with each other through memory. Conventional virtual memory hardware provides this feature for applications running in different address spaces, but some domains (*e.g.*, extensible type systems, extensible operating systems, and extensible applications) will not abide the high cost of multi-address-space inter-process communication. *Software-based* fault isolation allows multiple software modules to safely share a single address space [32], allowing low-cost, inter-module communication. To guarantee safety, modules monitor their own memory accesses in order to isolate themselves from one another.

Software-based fault isolation is motivated, described, and evaluated by Wahbe *et al.* [32]. We sketch the basic approach here. Each potentially unsafe instruction (*i.e.*, load, store, or branch) must be preceded by instructions to test that the module is permitted to load from, store to, or branch to the specified address. This code is inserted or verified by a trusted tool before execution. A check simply examines high-order address bits, which name the *segment* containing the address (analogous to a virtual memory page). Each module is permitted to access data in a single data segment and branch to code in a single code segment. A sample instruction sequence performing software-based fault isolation for a store appears in Figure 3.

The DISE equivalent of software-based fault isolation is straightforward. A production is required for each class of unsafe instruction (loads, stores, and branches). The replacement sequences look much like the code in Figure 3. For example, Figure 1(a) shows the DISE production for store instructions. Note that the replacement instructions use a number of DISE registers rather than architected registers, so the DISE implementation does not use registers that the principal computation would use otherwise. Software-based fault isolation, on the other hand, requires as many as five dedicated registers [32].

4.2 Path Profiling

Path profiling dynamically records the number of times each acyclic path in a program is traversed. Path profiles are used to identify and optimize the most frequently executed paths [3] and to evaluate test coverage [25, 26].

There are many approaches to computing or approximating path profiles. The simplest approach associates a tag combining a PC and the dynamic branch history leading to it with each path. At the end of the path (*i.e.*, at a function

```
addi $data-addr-reg,$sp,0          # data-addr-reg gets address
srl  $scratch,$data-addr-reg,25    # extract segment identifier
bne  $scratch,$segment-reg, err    # trap if bad segment identifier
sw   $r1,0($data-addr-reg)         # do store using data-addr-reg
```

Figure 3: Software-based fault isolation for the instruction “`sw $r1,0($sp)`.”

```
sll  $shift-reg, $shift-reg, 1    # make space for new branch
bne  $r2,$r3,target               # original conditional branch
ori  $shift-reg, $shift-reg, 1    # indicate branch not taken
```

Figure 4: Path profiling code for conditional branch “`bne $r2,$r3,target`.”

return or a loop back-edge), a counter associated with this tag is incremented. A post-execution pass reconstructs the paths from the tags. A simple formulation of this algorithm uses both a lossy (non-chaining) hash table, and a lossy (fixed-size) branch history. Information loss due to these simplifications is often minor and benign, because profile consumers usually do not require complete information. If necessary, greater accuracy may be achieved at higher cost (*e.g.*, longer branch history, larger or chained hash table).

A software-only implementation of the above formulation is realized as follows. Dynamic branch history is recorded via *bit tracing* in a shift register [6]. The outcomes of conditional branches are pushed onto this register. The shift register is saved and restored (using a stack) across procedure calls. At procedure returns and loop back-edges, we construct a tag from the current PC and the shift register and use it to index the *path-frequency hash table*. If we find this tag in the table, we increment its associated counter. Otherwise, we overwrite the table entry with the current tag and reset the counter. A software-only implementation of conditional branch shift register manipulation appears in Figure 4.[1] Before the branch, a zero is shifted into the branch history in `$shift-reg`, and if the branch is not taken, the zero is replaced by a one. Thus a zero indicates the branch is taken, and a one indicates that it is not taken.

We cannot simply form a replacement sequence of the instructions in Figure 4 for a DISE implementation. If we did, the instruction following the branch would always be executed if the branch was correctly predicted, independent of whether the branch was actually taken or not (details of this behavior are in Section 3). As a result, our DISE implementation introduces an intra-replacement-sequence branch as illustrated in Figure 1(b). The first three instructions of the sequence correctly update the shift register, while the final instruction actually transfers control. The DISE solution suffers from a longer instruction sequence (*i.e.*, four versus three instructions), which we evaluate in Section 5.3.

Like memory fault isolation, the DISE implementation benefits from the extra registers available to replacement sequence instructions. Path profiling requires at least two dedicated registers—the shift register and a shift register stack pointer—which a software-only implementation must steal from user code. In addition, some of the path profiling instruction sequences (see Appendix) are quite long, increasing code size. The use of procedure calls alleviates this problem, but the call itself adds dynamic instruction overhead. A DISE implementation inlines lengthy sequences with neither instruction footprint impact nor call overhead. DISE also has the benefit that sampled profiling may be trivially achieved by simply enabling and disabling the DISE facility. On the other hand, software-only path profiling using global control-flow analysis to implement counter-based rather that shift-register-based encodings can be more efficient [5]. A DISE implementation of this more sophisticated algorithm is impossible. However, DISE can be used in concert with static transformation, which we illustrate in the discussion of dynamic code decompression.

4.3 Dynamic Code Decompression

Code size is an important concern in embedded and mobile systems where strict cost constraints dictate small memories and caches. Static code compression coupled with a dynamic mechanism for decompression address this problem. Dynamic decompression systems come in two varieties. Decompressors that sit on the instruction cache miss path [17, 33] enable compressed memory images. Those that decompress the fetched instruction stream [21] also enable a compressed instruction cache footprint. Even high-performance processors may benefit from the second form

[1] The path-frequency hash table update code appears in the Appendix.

of decompression, as reduced cache footprints may enable the construction of smaller caches that can be accessed in fewer pipeline stages.

DISE implements post-fetch decompression by transforming fetched instructions into longer sequences of instructions, without instruction set redesign and without decompression-specific hardware. Counterintuitively, DISE potentially allows more sophisticated compression than that supported by dedicated decompressors. DISE's macro-expansion mechanism allows parameterized decompression (*i.e.*, a single decompressed code template may yield decompressed sequences with different register names when instantiated with different arguments). DISE also allows compression to be customized to a particular application, or even application kernel, by simple reloading of the PT and RT.

DISE compression/decompression consists of three steps: (i) computing a set of compression productions, (ii) transforming the executable so that compressible sequences are replaced by their triggers, and (iii) loading the productions into the PT and RT. We explain the last step first.

Our DISE decompression implementation uses reserved (*i.e.*, unused) opcodes to indicate compressed instructions (active opcodes cannot be used as they are needed for uncompressed instructions). Each production allows three register parameters. The remaining 11 bit immediate field (6 bits are taken by the opcode, and 5 each by the register parameters) could be used to "match" up to 2048 decompression sequences. However, such a formulation is impractical since the PT is a multi-ported fully associative table that cannot practically have more than 64 (or maybe even 32) entries. Decompression exploits an alternative PT usage model in which the 11 bit immediate field is interpreted as a concatenation of RT offset and length, rather than as a match input. This "trick" allows us to implement decompression using a single PT entry. We divide the 11 bit immediate field into an 8-bit offset and a 3-bit length, allowing us to use up to 256 decompression productions of up to 7 instructions each.

Our compression methodology constructs a set of productions customized for each application. First, we perform static analysis on the program to identify all possible instruction sequences of length 2-7. Sequences do not straddle basic blocks and do not contain system calls. Next, we combine *congruent* instruction sequences into a single *candidate replacement sequence*. Congruent sequences are identical except in three or fewer of their register operands. The identical register names are hard-coded into the candidate replacement sequence, and the differing registers are parameterized and will be dynamically instantiated by DISE.

Finally, we sort the candidate replacement sequences based on cumulative compression benefit. The sequence that results in the single largest benefit is first. The next sequence is the one that adds the largest additional benefit to its predecessors. This greedy process continues until all candidate replacement sequences have been considered. The compression factor used to compute the benefit of each sequence can be static (*i.e.*, proportional to the number of times the sequence appears in the static image) or dynamic (*i.e.*, proportional to the number of times a sequence is estimated to appear in the dynamic instruction stream), allowing us to target either static code size or reduced fetch. The replacement sequences we actually use for compression are a prefix of the sorted list of candidates.

Given a set of productions, compressing the program is straightforward.

4.4 Other Meta-Feature Implementations

Software fine-grain distributed shared memory. Shared memory systems provide the abstraction of a single shared address space among processors with physically distributed memory. Although software distributed shared memory can leverage the virtual memory hardware, this approach is limited because the granularity of sharing is tied to the size of the virtual memory page. To achieve fine-grain sharing, an application monitors each memory operation to determine whether it refers to private or shared data and whether shared data is present or not (as in Shasta [29]). DISE productions for these checks are similar to those used for memory fault isolation. Thus a DISE-capable machine can be configured to have the appearance of hardware-supported distributed shared memory without custom hardware.

Debugging and code assertions. Debugging is a central part of the code development process and code assertions are an invaluable part of debugging. Modern processors typically support limited hardware memory watchpoints allowing writes to a few memory locations to be monitored without undue overhead or changes to the debugged program. However, more complex assertions that involve the evaluation of arbitrary criteria are more cumbersome. Inserting these insertions into a program binary is impractical because they are very frequently changed during a single debugging session, so debuggers typically implement complex assertions by stepping through the program one instruction at a time and executing the assertion from the debugger itself. This process is extremely slow because the debugger is usually running in another process; even if it is not, instruction serialization neutralizes the parallelism

and pipelining capabilities of the underlying processor. DISE can be used to implement debugging assertions more efficiently. Assertions can be inlined into the program at arbitrary granularities and their execution interleaved with the original code without any serialization. Assertions can be added and removed quickly and even conditionally activated and deactivated automatically (*e.g.*, once an assertion fires, it is replaced by a second assertion). Inactive assertions have no runtime overhead.

5 Performance Evaluation

We experimentally evaluate the performance of DISE-implemented meta-features. The next subsection describes our experimental apparatus. The remaining subsections evaluate the three meta-features from the previous section.

5.1 Apparatus

We have modified the Simplescalar 3.0 timing simulator [7] to support DISE. Our baseline configuration models a 4-way superscalar processor with an eight stage pipeline and a maximum of 128 instructions or 64 memory operations in flight. We model a 1K-entry hybrid branch predictor with 2K-entry BTB. Our memory system consists of 32KB, 32 byte, 2-way set associative, 1-cycle-access instruction and data caches, ideal 1-cycle access TLBs, and an ideal, 12-cycle-access, unified L2 cache.

Our ten benchmarks come from the the SPEC CPU2000 integer benchmark suite (*eon* and *crafty* are incompatible with the PISA port of the gcc compiler). The benchmarks are compiled with gcc version 2.6.3 at optimization level -O3 and run to completion on the SPEC “test” inputs. We produce software-only meta-feature implementations via a script that performs simple textual transformations on the gcc-generated assembly code files. In order to avoid register re-allocation as part of this process, we modify gcc to use a subset of the available registers, so that the remainder may be used by the memory fault isolation or profiling code.[2] Note that this artificially reduces the number of registers available to the original program when it is possible for the principal and meta-feature instructions to share registers. The potential overlap is small (*i.e.*, at most one register) for the meta-features we examine.

The focus of this paper is to present DISE as a technique. More applications need to be studied before the DISE interface can be firmly defined. Here, we model a 32-entry PT, a 512-entry RT, and 16 dedicated DISE registers. Memory fault isolation and path profiling require (at most) four PT entries, twenty RT entries, and six dedicated registers. Dynamic code decompression requires 1 PT entry and uses as many RT entries as are available. In our decompression experiments, we use 128 replacement sequences.

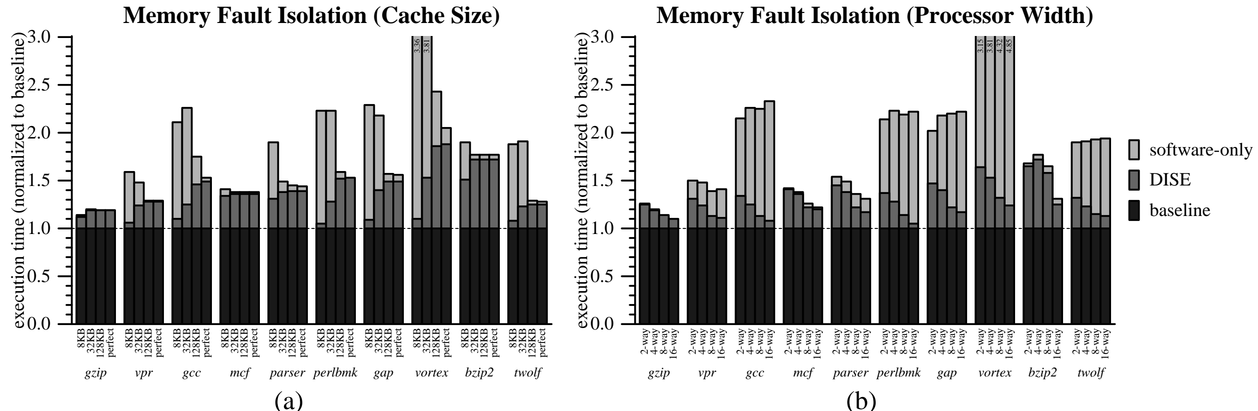
5.2 Memory Fault Isolation

The graphs in Figure 5 compare the performance of DISE and software-only implementations of memory fault isolation. There is a bar group for each benchmark and a bar for each of several different machine configurations (*e.g.*, varying instruction cache size in the left graph and varying processor width in the right graph). Our baseline machine is a 4-way superscalar with a 32KB instruction cache—the second bar in each group in each graph corresponds to this configuration.

The bars represent execution time normalized to that of the meta-feature-free baseline. Each bar is actually three overlaid bars corresponding to the execution times of the baseline (dark gray; always 1 since it is normalized to itself), DISE-implemented memory fault isolation (gray), and the software-only implementation (light gray). The bars are “depth-sorted,” allowing all three to be shown. If the light gray portion is on top of the stack of bars, the software-only implementation has the longest execution time. If the gray portion is on top, then the DISE implementation is slowest. DISE outperforms the software-only implementation of memory fault isolation in all benchmarks/configurations.

The runtime cost of a meta-feature implementations has two components. *Dynamic overhead* increases the dynamic instruction count, reducing effective pipeline bandwidth. *Static overhead* increases the static instruction working set, reducing the effective capacities of the static instruction-based structures such as the instruction cache, TLB, and branch predictor. Meta-features implemented entirely in software suffer from both static and dynamic overhead. DISE does not perturb the instruction working set, thus its meta-feature implementations suffer only from dynamic overhead. The two graphs attempt to isolate each of these forms of overhead.

[2] A temporary limitation of our approach to realizing software-only meta-feature implementations has prevented us from transforming *gcc* for path profiling, so no software-only path profiling *gcc* data is presented in this draft.



(a)

(b)

Figure 5: Performance of memory fault isolation.

Static overhead. Figure 5(a) shows the impact of static overhead, due primarily to degraded instruction cache performance. The performance overhead due to extra misses is greatest for benchmarks with high *a priori* miss rates. For 32KB caches, this includes *gcc*, *perlbmk*, *gap*, and *vortex*. At smaller cache sizes (8KB), miss rates and static overhead are significant for all but three of the benchmarks (*gzip*, *mcf*, and *bzip2*). As cache size increases, the static overhead of the software implementation decreases and the dynamic overhead remains constant. As a result the relative total overhead decreases. The DISE implementation does not have any static overhead, so the relative dynamic overhead grows as the baseline performance improves with the growing cache size.

Intuitively, the performance advantage of DISE decreases as cache size increases. As cache size grows, the static cost of the additional meta-feature instructions decreases. With a perfect cache, only the dynamic cost remains and the software-only approach and DISE become equal (from a runtime cost standpoint only; DISE still maintains its other advantages). These trends favor DISE, because (1) caches are not getting significantly larger, rather they are effectively shrinking as programs and their instruction working sets grow, and (2) cache size is limited by access latency.

Dynamic overhead. Figure 5(b) shows the impact of dynamic overhead. We fix the instruction cache size at 32KB and vary processor width from 2 to 16. As processor width increases, the dynamic (*i.e.*, bandwidth) overhead of both meta-feature implementations decreases. At high widths, data dependences limit parallelism within a fixed reordering window. If the width is increased beyond the ability of the principal application to utilize it, additional meta-feature instructions can exploit these idle functional resources at no perceived cost. The dispatch bandwidth used by additional meta-feature instructions is also cheaper at high processor widths. Decoding bandwidth utilization decreases as the front end is unable to sustain high fetch rates due to branch mispredictions and branch prediction bandwidth limitations. Not subject to fetch limitations, meta-feature instructions can easily utilize this unused decoding bandwidth.

As increased processor width drives down the dynamic cost of meta-feature instructions, the static cost is all that remains. In fact, this static cost becomes relatively higher. As the absolute cost of the principal application shrinks, the relative cost of each instruction cache miss grows. The *vortex* benchmark in Figure 5(b) underscores this point. This trend also bodes well for DISE: its advantages over software-only implementations will increase as processor performance grows.

5.3 Path Profiling

In the previous section, we isolated DISE's advantages over software-only implementations in terms of dynamic and static overhead, by varying instruction cache size and processor width, respectively. Here, we measure DISE's advantages along another dimension—fine-grain meta-feature toggling. Typically, profiling information need not be collected over a complete run of a program; small samples suffice. In fact, this profiling model is required by dynamic optimizing compilers [4, 11] which profile a running program for a short period, optimize based on that profiling information, and reap the performance benefits for the remainder of execution. Figure 6 compares the performance of software-only and DISE implementations of path profiling for meta-feature sampling granularities of 100% (*i.e.*, path profiling is always enabled), 50%, and 10%.

With continuous profiling (the 100% bar), the software-only implementation outperforms DISE for a number of

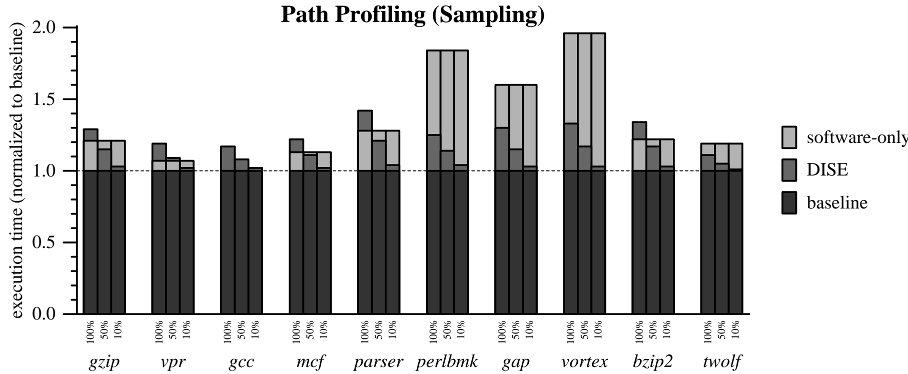


Figure 6: Performance of path profiling.

benchmarks (*gzip*, *vpr*, *mcf*, *parser*, and *bzip2*). DISE has a higher dynamic overhead, because its code for pushing each branch outcome onto the shift register has one more instruction than the corresponding software-only implementation. Benchmarks with small instruction working sets have nearly perfect instruction cache hit rates, even with the additional meta-feature instructions, and therefore have almost zero static overhead. The difference in dynamic overhead gives the software-only implementation an advantage for these applications. For the remaining benchmarks, the static overhead reduction of DISE compensates for its higher dynamic overhead.

While DISE is competitive in raw performance, it has an important dynamic advantage. DISE allows profiling overhead to decrease with the sampling rate—at a 10% rate, overhead is 2-3%. Software-only path profiling, on the other hand, cannot be enabled and disabled quickly, so its overhead is constant. Of course, not all meta-features can exploit sampling, but many, such as profiling and debugging support, can benefit from it.

5.4 Dynamic Code Decompression

Unlike memory fault isolation and path profiling, dynamic code decompression does not add functionality to a principal application. As there is no software-only counterpart to dynamic code compression/decompression, the graphs in Figure 7 characterize its DISE performance in isolation.

As described in Section 4.3, DISE compression may be tuned to optimize either static code size or dynamic fetch count. Figure 7(a) shows the reduction in static code size for both targets, represented by two overlaid, depth-sorted bars. Figure 7(b) shows reductions in the number of instructions fetched. Not surprisingly, optimizing compression for code size yields better code compression (some applications are compressed to 75% their original size), while fetch-count-driven compression yields lower dynamic fetch counts. Compression yields performance improvements of 4-25% when optimized for fetch count reduction (see the first bar of each group in Figure 7(c)) and 1-13% when optimized for static code size. DISE's programmability allows either goal to be satisfied on the same device.

Compression enabled hardware simplifications. Compression/decompression may be used to improve performance. However, it may also be used to achieve constant or near constant performance at reduced implementation cost. Figure 7(c) shows how the performance of fetch-count-optimized DISE compression can be traded for a smaller instruction cache, narrower fetch width, or both. The bars show execution time normalized to our base configuration: a 32KB 2-way set associative cache, 4-wide fetch, and no compression. The gray portion of each bar shows performance of uncompressed code. The light gray portion shows the performance of DISE compression/decompression.

From left to right, the bars in each benchmark group show our base configuration, a 16KB cache configuration, a 2-wide fetch configuration, and a configuration with both reduced fetch bandwidth and a smaller cache. All bars within a group are normalized to the same value (uncompressed execution time on the base configuration), so we can assess the relative impact of the simplified configurations. Reducing the cache to 16KB (second bar in each group) significantly raises execution times without compression (dark bars) for the benchmarks most sensitive to cache size (*gcc*, *perlbmk*, *gap*, *vortex*, and *twolf*). With compression (light bar), the smaller cache does not significantly degrade performance for any of the benchmarks. When the fetch bandwidth is reduced from four to two wide (third bar in each group), the uncompressed execution time increases dramatically for all benchmarks, because the fetch unit is unable to keep the four-wide machine fed with instructions. Compressed execution time is much less sensitive to this

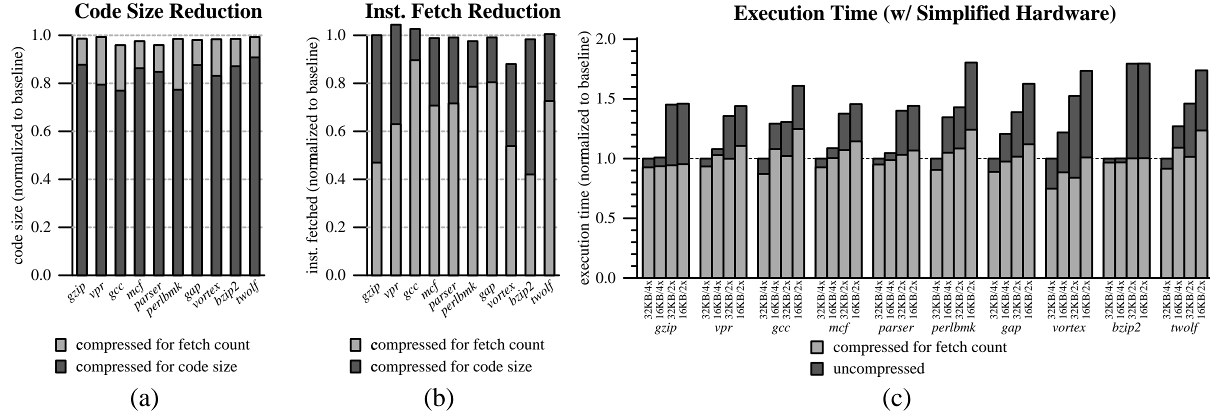


Figure 7: Code compression/decompression statistics.

reduction, because the decoder is injecting instructions into the instruction stream. The final bar shows execution times for a machine with both a smaller cache and reduced fetch bandwidth. The compressed code on the smaller machine configurations has performance competitive with (often better than) the uncompressed code on the base machine configuration. Compression enables reduced hardware implementation without significant performance loss.

Note, our reduced cache results do not account for the possibility that a smaller cache could be accessed in fewer cycles. DISE performance would improve further if that were the case.

6 Conclusion

Dynamic instruction stream editing (DISE) is a cooperative software-hardware mechanism that efficiently adds meta-features to applications via programmable decoding. DISE is a single, general facility capable of realizing a large class of meta-features, three of which (memory fault isolation, path profiling, and dynamic code decompression) we have implemented and evaluated with our DISE simulator. Furthermore, adding meta-features to applications via DISE makes efficient use of resources, has no cost associated with transforming the binary, and is highly dynamic in that translation behavior can change during application execution. The implementation of the DISE facility itself requires only modest, isolated changes to the processor decoder.

Our experiments show that DISE often has significantly better performance than software-only implementations of meta-features and that architectural trends suggest that the value of DISE will only increase with time. We also demonstrate effective code compression with DISE.

We find that DISE is a promising facility with many potential applications. We intend to study the DISE implementation of other meta-features to refine our design and further quantify its benefits and limitations.

References

[1] A. Agarwal, R. Sites, and M. Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proc. 13th International Symposium on Computer Architecture*, pages 119–127, May 1986.

[2] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, 1999.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dyanmo: A transparent dynamic optimization system. In *Proc. of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dyanmo: A transparent dynamic optimization system. In *Proc. of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[5] Thomas Ball and James Larus. Efficient path profiling. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[6] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[7] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin–Madison Computer Sciences Department, 1997.

[8] R. E. Calcagni and W. Sherwood. Patchable control store for reduced microcode risk in a VLSI VAX microcomputer. In *Proc. of the 17th Microprogramming Workshop*, pages 70–76, 1984.

[9] Yuan C. Chou and John Paul Shen. Instruction path copressors. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 270–281, 2000.

[10] K. Diefendorf. K7 challenges Intel. *Microprocessor Report*, 12(14), November 1998.

[11] E. Duesterwald and V. Bala. Software profiling for hot path prediction. In *Proc. the 9th Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[12] K. Ebcioglu and E. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proc. 24th International Symposium on Computer Architecture*, pages 26–38, Jun. 1997.

[13] P. Glaskowsky. Pentium 4 (partially) previewed. *Microprocessor Report*, 14(8), August 2000.

[14] L. Gwenapp. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2), February 1995.

[15] L. Gwenapp. Nx686 Goes Toe-to-Toe with Pentium Pro. *Microprocessor Report*, 14(9), Oct. 1995.

[16] L. Gwenapp. P6 Microcode can be Patched. *Microprocessor Report*, 11(12), Sept. 1997.

[17] IBM. *CodePack PowerPC Code Compression Utility User's Manual Version 3.0*. IBM, 1998.

[18] I. Kim and M. Lipasti. Implementing Optimizations at Decode Time. In *Proc. 29th International Symposium on Computer Architecture*, pages 221–232, May 2002.

[19] Alexander Klaiber. The technology behind Crusoe processors. Transmeta Corporation White Paper, January 2000.

[20] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proc. of the 1995 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 1995.

[21] C. Lefurgy, P. Bird, I.-C. Cheng, and T. Mudge. Improving Code Density Using Compression Techniques. In *Proc. 30th International Symposium on Microarchitecture*, pages 194–203, Dec. 1997.

[22] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, 1995.

[23] R. Nair and M. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proc. 24th International Symposium on Computer Architecture*, pages 13–25, Jun. 1997.

[24] T. Rauscher and A. Argawala. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers*, C-27(11):1006–1014, 1978.

[25] T. Reps. The Use of Program Profiling in Software Testing. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik'97*. Springer-Verlag, Sep. 1997.

[26] T. Reps, T. Ball, M. Das, and J. Larus. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *Proc. 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sep. 1997.

[27] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proc. of the USENIX Windows NT Workshop*, August 1997.

[28] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4), November 1997.

[29] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[30] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[31] J. Turley. Alpha Runs X86 Code with FX!32. *Microprocessor Report*, 10(3), Mar. 1996.

[32] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[33] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 81–91, 1992.

[34] C.B. Zilles and G.S. Sohi. A programmable co-processor for profiling. In *Proc. 7th International Symposium on High Performance Computer Architecture*, 2001.

A Additional Path Profiling Code

The code sequences in Figure 8 are inlined into the program binary when path profiling is implemented in software, and they are defined as replacement sequences for a DISE implementation. In the latter case, the directives of all but the last instruction of (b) and (c) are literal (the jump targets are taken from the trigger instruction). The initialization instructions in (a) allocate the path frequency hash table and the shift register stack and initialize the hash table and dedicated register state (`$hbase` and `$dsp`). The operating system executes a single trigger for this sequence on behalf of the application. The instructions for function calls in (b) simply push the shift register onto the path stack and then clear it. The sequence for function returns in (c) records in the hash table the path leading to the return. Lines 1–6 hash the program counter and shift register (`$dsr`) to index into the hash table (at address `$hbase`). Lines 7 and 8 read the tag and its counter from the indexed entry in the hash table. Lines 9–11 increment the counter or reset it if the tags do not match. Lines 12 and 13 store the tag and counter back to the table. Lines 14 and 15 restore the previous shift register from the path stack. And the final line actually performs the function return. A similar sequence—less the path stack manipulation—is used for loop back edges.

```
 1   li      $a0,#HT_SIZE
 2   djal    malloc                 # allocate hash table
 3   addi    $hbase, $v0, 0         # place base addr in dedicated register
 4   addi    $a0, $v0, 0
 5   djal    init_hashtab           # initialize hash table
 6   li      $a0,#STACK_SIZE
 7   djal    malloc                 # allocate shift-reg stack
 8   addi    $dsp, $v0, #STACK_SIZE  # place top-of-stack addr in dedicated reg
```

(a)

```
 1   sw     $dsr, 0($dsp)
 2   subi   $dsp, $dsp, #4          # push shift register onto path stack
 3   sub    $dsr, $dsr, $dsr        # clear shift register
 4   jal    target                  # call original target
```

(b)

```
 1   lui    $tag, $pc              # hi(tag) = PC
 2   andi   $dsr, $dsr, 0xffff
 3   or     $tag, $tag, $dsr       # low(tag) = path
 4   sll    $hidx, $tag, #HT_SHIFT # compute index into table
 5   andi   $hidx, $hidx, #HT_MASK # compute index into table
 6   add    $hidx, $hidx, $hbase   # compute entry addr in table
 7   lw     $htag, 0($hidx)        # load tag
 8   lw     $hcnt, 4($hidx)        # load counter
 9   dbeq   $htag, $tag            # compare tag == generated tag?
10   li     $hcnt, 0               # no, clear counter
11   addi   $hcnt, $hcnt, 1        # inc counter
12   sw     $hcnt, 4($hidx)        # store counter
13   sw     $tag, 0($hidx)         # store tag
14   addi   $dsp, $dsp, 4          # update path stack
15   lw     $dsr, 0($dsp)          # pop previous shift reg
16   j      $ra
```

(c)

Figure 8: Path profiling code for (a) initialization, (b) function calls, and (c) function returns.