DISE: A Programmable Macro Engine for Customizing Applications

Marc L. Corliss, E Christopher Lewis and Amir Roth
Department of Computer and Information Science
University of Pennsylvania
{mcorliss,lewis,amir}@cis.upenn.edu

Abstract

Dynamic Instruction Stream Editing (DISE) *is a cooperative software-hardware scheme for efficiently adding customization functionality—e.g, safety/security checking, profiling, dynamic code decompression, and dynamic optimization—to an application. In DISE, application customization functions (ACFs) are formulated as rules for macro-expanding certain instructions into parameterized instruction sequences. The processor executes the rules on the fetched instructions, feeding the execution engine an instruction stream that contains ACF code. Dynamic instruction macro-expansion is widely used in many of today's processors to convert a complex ISA to an easier-to-execute, finer-grained internal form. DISE co-opts this technology and adds a programming interface to it.*

DISE unifies the implementation of a large class of ACFs that would otherwise require either special-purpose hardware widgets or static binary rewriting. We show DISE implementations of two ACFs—memory fault isolation and dynamic code decompression—and their composition. Simulation shows that DISE ACFs have better performance than their software counterparts, and more flexibility (which sometimes translates into performance) than hardware implementations.

1. Introduction

The diversification of computing platforms (server, workstation, laptop, handheld, etc.) has increased the role of ***application customization functions (ACFs)***, utilities that customize the execution of an application for a particular environment. Examples of ACFs include safety checking (mobile code), profiling and dynamic optimization (high performance processors), dynamic code decompression (embedded processors), and bug-patching in the field (unfortunately, all processors). Considerable research effort has been devoted to implementing ACFs in both hardware—on dedicated, potentially programmable pipeline stages—and software—by embedding ACF code into an application as additional instructions, often via binary rewriting. The two approaches have complementary strengths and weaknesses. Hardware implementations contribute little or no overhead to the application but are functionally rigid. Even programmable designs are restricted by the number of stages, their positioning in the pipeline, and the basic operations they perform. Software implementations are functionally rich but degrade application performance. The additional instructions effectively reduce both instruction cache capacity and pipeline throughput. In addition, the binary rewriting process itself exacts a fixed cost that constrains the granularity at which ACFs can be profitably used.

Dynamic instruction stream editing (DISE) is a collaborative software-hardware approach to implementing ACFs, one that combines the flexibility of software implementations with some of the performance benefits of hardware ones. In DISE, ACFs are formulated as dynamic instruction stream transformations, sets of rules (macros) for replacing instructions that match certain criteria with parameterized instruction sequences. The processor executes the rules on the application's fetch stream, feeding the execution engine an instruction stream that includes ACF code. Code insertion prior to execution enables ACFs that modify, not just observe, application behavior and allows the processor to dynamically allocate execution resources between ACF and application code. Code insertion after fetch sidesteps many of the costs of statically embedding ACF code into the application's static image. Using instructions to implement ACFs is general and eliminates the need for specialized hardware widgets.

DISE is a single facility that unifies two classes of ACFs. ***Transparent ACFs*** act on unmodified "out-of-the-box" or "off-the-wire" executables by redefining the semantics of "naturally occurring" instructions. ***Aware ACFs*** act on modified applications into which specially constructed codewords have been planted by a DISE-aware static rewriting tool; the semantics of these codewords is defined by the DISE rules. In this paper, we show DISE formulations of two ACFs. ***Memory fault-isolation*** is a transparent ACF that is implemented by defining fault detection replacement rules for loads, stores, and indirect jumps. ***Dynamic code (de)compression*** is an aware ACF that is implemented by defining expansions for frequently occurring code sequences. The binary is compressed by replacing instances of those sequences with DISE codewords. At runtime, DISE expands the codewords, recreating the original instruction stream. We also show dynamic composition of ACFs via DISE.

We evaluate DISE using cycle-level simulation of the SPEC2000 integer benchmarks. Our results show that DISE memory fault isolation degrades application performance less than the corresponding binary rewriting implementations (and this is without accounting for the initial cost of the rewriting process itself). DISE code (de)compression yields compression factors and speedups competitive with those possible via dedicated decompressors, and without decompression specific hardware.

Decoder-based instruction macro-expansion is used in many CISC (e.g., IA32) processors to present the execution engine with a finer-granularity, easier-to-execute RISC-like instruction stream [10, 12, 14, 15]. DISE perform a logically different function—it adds functionality to an executing program by inserting additional ISA instructions—but uses similar hardware mechanisms. DISE co-opts this technology and adds a programming

Appears in *Proceedings of the Thirtieth International Symposium on Computer Architecture*, June 2003.

interface to it. In processors that already contain CISC-to-RISC decoders, DISE may also co-opt the physical structures.

The rest of the paper is organized as follows. Section 2 presents the DISE mechanism itself and Section 3 describes DISE ACF implementations. Section 4 contains a performance evaluation. The final sections summarize related work and conclude.

2. DISE

DISE inspects every fetched instruction and macro-expands those that match specified criteria. We call the macro-expansion rules ***productions***. A production is composed of a ***pattern specification*** and a ***replacement sequence specification***. A fetched instruction that matches a pattern is called a ***trigger***. DISE replaces the trigger with a ***replacement sequence*** which is formed by combining the replacement sequence specification with information (i.e., bits) from the trigger.

DISE has two major components. The ***engine*** applies the productions to the fetch stream. The ***controller*** implements the interface between the DISE engine and the rest of the system. This section describes the DISE engine and controller. We focus primarily on functionality. However, since the engine is performance critical—it inspects every fetched instruction—we dedicate a subsection to exploring its implementation. At this time, we are less concerned with the implementation and performance of the controller which is invoked only when the productions themselves are manipulated (i.e., rarely).

2.1. Engine Functionality

The DISE engine is a native-to-native expander that transforms the instruction stream in peephole fashion, one instruction at a time, each expansion physically independent of the rest. However, even with these restrictions—which simplify the programming model and the hardware implementation—DISE supports the formulation of many interesting ACFs. Three features help in this regard: ***parameterized replacement***, a ***dedicated register space***, and ***replacement-sequence control flow***. The utility of parameterized replacement is obvious. The latter two allow individual replacement sequences to perform complex tasks and to synthesize global behavior by linking independent expansions via persistent register communication. They also relieve two headaches associated with software ACF implementations: scavenging registers from the application and retargeting application branches around inserted ACF fragments.

Matching and replacement. DISE's basic operation is instruction pattern matching and parameterized replacement. A pattern specification may include any combination of opcode, opcode class, logical register names, immediate field or attribute thereof (e.g., its sign). For example, DISE is able to specify patterns of the form ''loads that use the stack-pointer as their address register'' or ''conditional branches with negative offsets.'' Currently, patterns are only defined on instruction bits. We leave open the possibility of matching other attributes (e.g., PC).

To enable interesting ACFs, replacement sequences are parameterized. Each replacement instruction field comes with a directive that (optionally) instantiates it using a field from the trigger. Different fields have different directives. For instance, register fields have five possible directives: **literal**, **dedicated**, **T.RS**, **T.RT**, and **T.RD**. The literal directive means the register number is to be interpreted literally. The dedicated directive means that the register number is one of the DISE dedicated registers (see next sub-heading). The last three directives cause the register number to be replaced with one of three register numbers from the trigger. Opcode and immediate fields have analogous directives. Parameterization permits transformations like the following: "replace loads with a sequence of instructions that masks the upper bits of the address and then performs the original load." We *have* found uses for non-instruction attributes in replacement (e.g., the ability to encode the trigger's PC in a replacement instruction immediate is useful in profiling ACFs [8]).

Memory Fault Isolation (MFI)	Fetch Stream
P1: T.OPCLASS == store ->**R1**	stq a0, 8(t0)
P2: T.OPCLASS == load ->**R1**	**Execution Stream**
R1: srli **T.RS**, 26, $dr1	srli **t0**, 26, $dr1
cmpeq $dr1, $dr2, $dr1	cmpeq $dr1, $dr2, $dr1
bne $dr1, error	bne $dr1, error
T.INSN	**stq a0, 8(t0)**

FIGURE 1. Memory fault isolation in DISE

Figure 1 shows productions that implement memory fault isolation (described in Section 3.1). There are two pattern specifications, one matches loads, the other stores. Both pattern specifications are associated with the same replacement sequence specification, **R1**. The sequence extracts the segment (high-order) bits from the address register, checks that the segment is legal, branches to an error handler if it is not, and executes the original instruction otherwise. The full fault isolation implementation uses a third production to match all indirect jumps and replace them with a sequence similar to **R1** that compares the target register's segment bits to the application's legal text segment.

Parameterization is used in two places. In the first replacement instruction, we extract the segment bits from **T.RS**, the address register of the trigger. The final instruction is the original trigger itself, **T.INSN**. The remaining instruction components are literals (e.g., srli, 26) or dedicated registers (e.g., $dr1). The right side of Figure 1 shows a fetched store instruction and the replacement sequence that is executed in its place (**t0** replaces **T.RS**, the entire store replaces **T.INSN**).

Dedicated registers. Replacement instructions can access dedicated registers that are not accessible via application code. The dedicated register set provides individual expansions with temporary register storage without having to save and restore user registers. More importantly, it provides persistent register storage across expansions allowing global ACFs to be synthesized from local expansions, without requiring the compiler to reserve user registers for the communication. These properties simplify ACF formulations and reduce their runtime cost. For example, in Figure 1, $dr2 is a dedicated DISE register that contains the current application's legal data segment identifier. The ACF initializes this register, which is subsequently used in every replacement sequence. The application has no direct way to modify this register. $dr1 is used for temporary (scratch) storage.

Replacement sequence semantics. To simplify ACF pro-

gramming, we define the semantics of replacement sequences to resemble and complement those of conventional (i.e., application) instruction sequences. First, we define precise state at replacement instruction boundaries, allowing replacement sequences to use a standard interrupt model. Second, we provide a two-level control model that allows replacement sequences to include control transfers but requires each replacement sequence to appear to be fully contained within the trigger it replaces. In other words, control can be transferred either at the application instruction level (the conventional way) or completely within a single dynamic replacement sequence; one dynamic replacement sequence cannot jump into the middle of another.

We formalize these definitions via a state element called the ***DISEPC***, a program counter that acts at the replacement sequence level. Every dynamic instruction is tagged with a PC**:**DISEPC pair. For an application instruction, DISEPC is 0. For a replacement instruction, PC is the trigger's PC and DISEPC is its offset from the start of the replacement sequence.

Precise state is defined at each PC**:**DISEPC boundary. If instruction PC**:**DISEPC is interrupted, post-handler fetch restarts at PC**:**DISEPC. The fetch engine ignores the DISEPC component, fetching the application instruction at PC. The DISE engine recognizes the annotation and expands the replacement sequence starting at DISEPC, skipping the first DISEPC–1 instructions.

The PC**:**DISEPC pair also creates the two-level control model by allowing replacement instructions to change either the PC or DISEPC, but not both. Application branches transfer control at the application instruction level, change the PC only (their target DISEPC is implicitly 0), and are in general oblivious to the presence of DISE. For control transfers *within* a replacement sequence, DISE provides variants of branches and jumps that modify the DISEPC only. A replacement sequence may contain both application (conventional) and DISE (sequence internal) branches. DISE also allows replacement sequences to include function calls, a useful feature for implementing complex ACFs.

The semantics of application branches within replacement sequences are subtle. The basic question is whether post-branch replacement instructions belong to the branch's taken or non-taken path. The answer is neither; they belong to the branch's *predicted* path and are squashed if the branch is *mispredicted*. These semantics seem odd but follow a simple logic and mesh with the way superscalar control flow operates. There are two cases of interest. If the branch is the trigger—i.e., an application branch that was expanded into a sequence in which it was not the last instruction—then it was initially predicted and the replacement sequence instructions that follow it must be associated with its predicted path or else the correct-path application instructions that follow it will also need to be squashed. While allowing the contents of the instruction stream to depend on branch prediction is strange, we must remember that this behavior only occurs for productions that match branches and place them in the *middle* of the resulting replacement sequences. Such productions should not be used if this behavior is not desired. If the branch is not the trigger—it simply appears in a replacement sequence for some other instruction—then it was never predicted. Since a non-predicted branch is effectively predicted non-taken, the replacement instructions that follow it will be associated with the non-taken path and discarded if the branch is taken. This is often the desired behavior as exemplified by our memory fault isolation production in Figure 1: if the address check fails, the store is flushed and fetch resumes at the error handler.

Explicit tagging and DISE usage modes. Thus far, we showed fetched instruction bits used both for matching and parameterization. DISE also allows trigger bits that are not used for either matching or parameterization to be interpreted as replacement sequence identifiers (tags). Use of this feature—***explicit tagging***—classifies ACFs into two categories.

Transparent ACFs operate on unmodified executables using productions that match "naturally occurring" instructions. Examples of transparent functionality include memory fault isolation (productions are defined for loads, stores, and jumps as in Figure 1) and branch profiling (productions are defined for branches). Transparent productions do not use explicit tagging since the bits of "naturally occurring" instructions cannot be interpreted as replacement sequence identifiers. Here, the mapping from pattern to replacement sequence is set up offline.

Aware ACFs operate on modified applications into which specially crafted DISE codewords (instructions that do not occur naturally) have been planted. Code decompression is an aware ACF. A DISE-aware utility compresses the original executable by replacing common multi-instruction sequences with DISE triggers. At runtime, DISE replaces the codewords with the corresponding original sequences. One way of generating codewords is with reserved opcodes. Since the number of reserved opcodes is limited, aware ACFs can use explicit tagging to map a single opcode (via a single pattern) to multiple replacement sequences. For instance, in a 32-bit ISA with 6-bit opcodes and 5-bit register specifiers, an aware ACF using a single reserved opcode and 3 register parameters can use the remaining 11 bits to name 2048 replacement sequences, effectively creating 2048 distinct codewords.

2.2. Engine Implementation

A DISE engine implementation has three primary objectives: zero performance degradation on ACF-free code, ACF performance that equals or exceeds that of a corresponding software implementation, and minimal performance, design, and verification impact on the rest of the microarchitecture. We propose an implementation here and discuss some possible alternatives.

Basic structures. The instruction to instruction-sequence expansion performed by DISE is similar in spirit—and thus in implementation—to the instruction to *micro*-instruction-sequence expansion performed by many IA32 processors to convert complex instructions to a more regular (three register) internal form [10, 12, 14, 15].

Three structures are used. The ***pattern table (PT)*** contains pattern specifications. On architectures with a regular ISA encoding, matching may be performed by masking and comparing raw instruction bits. On architectures with irregular ISA encoding, instructions may need to be partially decoded and reformatted before PT access for more efficient matching. It is likely that such processors already contain pre-decode facilities. Logically, the PT is a fully-associative structure that matches every instruction to all active patterns. If multiple patterns match, the most specific one (the one that matches the greatest number of instruc-

tion bits) is chosen. This facility allows the construction of overlapping pattern specifications and even negative specifications. For instance, to specify the pattern "all loads that don't use the stack pointer", two patterns are used. One matches loads that use the stack pointer and performs the identity expansion. The other matches all loads and performs the desired task.

In addition to a pattern specification, each PT entry contains a replacement sequence identifier, which is either the identifier itself (for transparent productions) or a mask that delineates its position within the trigger (for aware productions).

The ***replacement table (RT)*** is a small cache that houses replacement sequences. Each RT entry corresponds to a single instruction from a replacement sequence specification and contains a replacement literal and a series of instantiation directives. Each entry is tagged by the replacement sequence identifier and the instruction's offset within the sequence (i.e., its DISEPC). The tag also contains the length of the replacement sequence in which the instruction is embedded; this field aids with RT virtualization. The RT may be direct-mapped or set-associative. Multiple sequential instruction specifications may be coalesced into a block, reducing the number of RT read ports at the expense of internal fragmentation.

The ***instantiation logic (IL)*** is a combinational circuit that executes instantiation directives to combine replacement literals with trigger fields and produce the actual replacement instructions that are spliced into the application's execution stream.

Pipeline organization. The contents and positioning of the PT and RT depend on existing decoder pipeline structure, and the presence of CISC-to-RISC macro-expansion facilities.

Logically, the PT and RT are accessed in series. While both can be made small, it is likely not the case that they can be read serially in a single cycle. If the existing decoder is implemented in multiple stages, the PT and RT my be positioned in series with no penalty. In a single stage decoder, however, this organization elongates the pipeline and violates our first design goal: zero performance impact on ACF-free code. An alternative is to position the PT and RT in parallel and incur a one cycle stall for each actual expansion. Figure 2 (top) illustrates this organization.

The presence of a CISC instruction to RISC micro-instruction macro-expansion facility raises the possibility of unifying the two mechanisms. Logically, DISE is an instruction to instruction sequence transformer that precedes instruction to micro-instruction conversion. While this logical organization can be mirrored physically—with DISE feeds an ACF-augmented CISC stream to an unmodified and completely microarchitectural CISC-to-RISC decoder—the two facilities are so similar structurally that they may be combined into a single complex. Figure 2 (bottom) illustrates this organization using the P6 decoder [14] as a rough guide. CISC to internal RISC translation is performed in two ways. Translation resulting in four or fewer micro-instructions is done via combinational logic arrays (CLAs). Translation requiring longer sequences is performed by sequentially instantiating templates from a ROM (µROM). CLA/µROM multiplexing (and µROM indexing) is based on the CISC opcode and performed by a selector, itself either a CLA or ROM. The PT and RT parallel the selector and µROM, respectively, in functionality and positioning. A PT match overrides both the µROM and CLA. The CISC-to-RISC and DISE ILs are physically unified. The RT

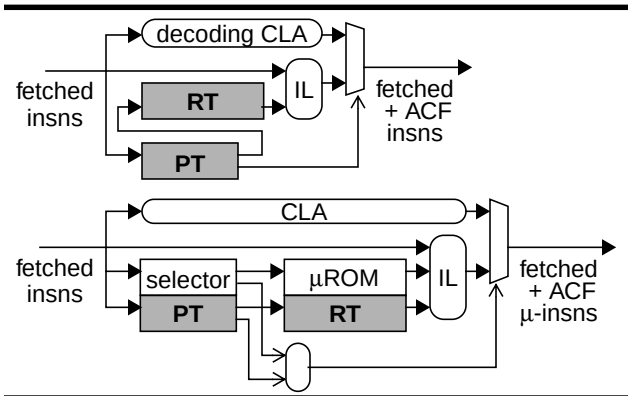

FIGURE 2. DISE engine implementations

and µROM may also be unified, although the original µROM contents must be kept immutable and invisible.

The unified organization requires the RT to contain replacement sequences in micro-instruction form. Since replacement sequences are externally specified at the instruction level, they must be translated before being placed in the RT. The translation is performed by the DISE controller on RT fills, and may use the existing CISC-to-RISC decoding path.

Control and DISEPC. Replacement sequence precise state and internal control are implemented using the DISEPC state element. In a DISE-enabled pipeline, every instance of the PC is expanded to a PC**:**DISEPC pair. Only the DISE engine interprets the DISEPC annotations—initiating expansions at the DISEPC instruction of a replacement sequence—all other stages ignore it. A fault at PC**:**DISEPC invokes the operating system, which saves the excepting PC**:**DISEPC as a pair. When the handler terminates, control returns to PC**:**DISEPC. Fetch ignores the DISEPC annotation, DISE recognizes it and expands the replacement sequence starting at offset DISEPC. DISE internal control is implemented in the same way. Since DISE branches are not predicted, a taken DISE branch is interpreted as a mis-prediction. Fetch is restarted at the same PC, but a new DISEPC. To obtain the right behavior for conventional branches embedded within replacement sequences we suppress the prediction of non-trigger replacement sequence branches. This is trivial if pre-decode information is used. If, on the other hand, the microarchitecture interprets BTB hits as branch indicators, then non-trigger replacement branches must be prevented from updating the BTB.

2.3. Interface and System Architecture

In addition to having a lightweight implementation, DISE must be flexible, portable and secure. These goals are achieved using two layers of abstraction and access control: PT/RT manipulation is mediated by a hardware controller and access to the controller is in turn mediated by the OS kernel. The controller abstracts the internal formats of the PT and RT and virtualizes their sizes. The OS kernel virtualizes the active set of productions, preserving transparency of multiprogramming and protecting applications from interference by foreign code. These two layers present the user with a simple interface for managing DISE productions. This interface may be instruction-based or memory-mapped and the productions themselves may reside in

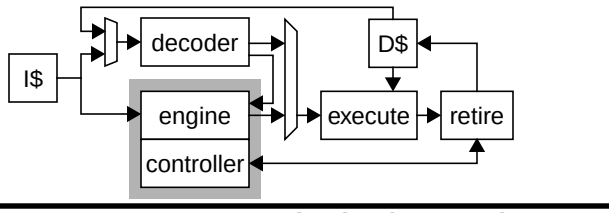

FIGURE 3. DISE engine implementations

either instruction or data space. Figure 3 diagrams an architecture with an instruction-based interface and productions residing in data space.

Controller. The controller provides an interface for programming the PT and RT, abstracting the internal formats of patterns and replacement instructions from DISE users. By owning the controller, the processor vendor retains the freedom to change internal formats and PT/RT implementation details in future products. The controller translates productions from their external representation—a directive-annotated version of the processor's native ISA—to the internal formats used by the PT/RT. As shown in Figure 3, the decoder may be used for this translation.

The controller also virtualizes the sizes of the PT and RT. For minimal impact on decoding latency, these structures—especially the PT which must be multi-ported—must be relatively small. However, restricting their sizes limits DISE's applicability. Virtualization—treating the PT/RT as physical caches for a larger virtual namespace—solves this problem. It also enhances the portability of DISE ACFs and reduces the amount of DISE state that must be maintained across context-switches. The main challenge of virtualization is capturing the notion of PT and RT misses. RT miss detection is easy. An RT miss is detected when an identifier/DISEPC pair generated by the PT is not in the RT. PT miss detection is more difficult as a missing entry is naturally interpreted as a non-match. To identify PT misses we track the number of active and PT-resident patterns associated with each opcode in a small direct-mapped table. A fetched instance of an opcode with differing active and resident pattern counters indicates a PT miss, triggering a fill of *all* patterns for that opcode. In this organization, the ***pattern counter table*** (which is of fixed virtual size) is the only piece of architectural state associated with the PT/RT complex; the contents of the PT/RT can be "faulted in." A PT or RT miss interrupts the processor via the controller. The mechanics of PT/RT miss handling resemble those of software TLB miss handling and have similar costs. The pipeline is flushed and the missing productions are loaded procedurally, again via the controller.

OS kernel. The OS kernel virtualizes the resident set of productions. The first aspect of this is the preservation of DISE state across context switches. DISE state consists of the contents of the pattern counter table, DISEPC and DISE registers; the PT and RT are demand loaded. The DISE registers and DISEPC are accessed via conventional datapaths, using either privileged instructions or save/restore replacement sequences. The pattern counter table is accessed via the controller.

The second aspect of production virtualization is the separation of production sets generated by and targeted for different applications from one another. Non-separated production sets give rise to security problems, with one process using DISE productions to read and write the state of a second process, perhaps even the OS kernel. One component of the solution is to force DISE users to submit ACFs to the kernel for "inspection and approval." However, this policy should not be applied universally, as some ACFs may rely on fast user-level access to the production interface. A compromise permits applications to access the DISE controller directly, but uses the OS kernel to control the contents of the PT/RT across context-switches. Productions that reside in an application's data space (i.e., they were not submitted via the kernel API) are limited to operating on that application only, they are deactivated when the application is switched out. Productions that are submitted to and approved by the kernel reside in kernel space and can operate on other applications. Transparent ACFs, many of which have a system-utility flavor and are supplied by the OS vendor, use this model. The submission API enables third-party utilities.

3. ACF Formulations in DISE

DISE enables the implementation of a large class of ACFs. In this section, we present implementations of two ACFs—memory fault isolation and dynamic code decompression. We also explain how ACFs can be dynamically composed. These ACFs are not new. Their unification via a single hardware facility is.

3.1. Transparent ACFs

Transparent ACFs are applied to unmodified binaries, augmenting or redefining the functionality of "naturally occurring" machine instructions. We describe several below.

Memory fault isolation. Memory fault isolation prevents multiple applications from interfering with one another through memory. Virtual memory provides this feature for applications running in different address spaces, but some domains (e.g., extensible operating systems and applications) cannot tolerate the high cost of multi-address-space inter-process communication. Software-based fault isolation [32] allows multiple modules to safely share a single address space by statically rewriting each module to monitor every memory reference (load, store, or indirect jump). There are two variants. *Sandboxing* precedes each unsafe instruction with a code sequence that sets its high-order address bits to the module's assigned code or data segment identifier. *Segment matching* precedes each unsafe instruction with an address check and jumps to an error routine if the check fails.

Both variants are easily implemented in DISE: the address checks/modifications are inlined dynamically rather than via static transformation. Figure 1 illustrates the segment matching case. The DISE formulation is actually computationally cheaper than the software one in two ways. First, the software implementation requires as many as five dedicated registers that must be reserved by the compiler or scavenged by a rewriting tool. Second, each software-inserted code fragment requires an extra copy instruction to ensure that malicious jumps into the middle of the sequence do not subvert the address check. The DISE implementation employs dedicated DISE registers and does not require additional copies since the DISE control-flow model disallows jumps into the middle of replacement sequences.

Other transparent ACFs. Path profiling [3] dynamically records the number of times each acyclic path in a program is tra-

versed. Path profiles are used to identify "hot" paths for optimization [2] and evaluate test coverage [24]. Path profiling associates a tag—e.g., PC and conditional branch outcome history—with each static path. At an acyclic path endpoint—function return or loop back-edge—a counter associated with this tag is incremented. A post-execution pass reconstructs paths from tags. As profile consumers usually do not require complete information, the counter maintenance scheme may be lossy. DISE allows a simple "bit tracing" implementation of profiling via productions for conditional branches and function returns. A complete description of a DISE path profiler is available here [8].

Distributed shared memory (DSM) provides the abstraction of a single shared address space among processors with physically distributed memory. Software DSM that leverages virtual memory hardware is limited to sharing at the page granularity. To achieve fine-grain sharing in software, 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 [28]). DISE productions for these checks are similar to those used for memory fault isolation. In this way, a DISE-capable machine can be configured to have the appearance of hardware-supported fine-grained DSM without custom hardware.

Code assertions are an invaluable part of debugging. Although modern processor support limited hardware memory watchpoints, more general assertions involving the evaluation of arbitrary criteria must be implemented in software. Debuggers typically implement complex assertions by single-stepping through the program, executing the assertion from the debugger itself. This process is extremely slow. Even if the debugger is not running in another process—it usually does—instruction serialization neutralizes the parallelism and pipelining capabilities of the underlying processor. With DISE, debugging assertions are inlined into the program at arbitrary granularities and their execution is interleaved with the original code without serialization. Assertions can be added and removed quickly. Inactive assertions have no runtime overhead.

Reference monitors implement security policies by observing program execution, terminating it if some policy is violated [26]. They may control an application's use of memory [32], library calls [33], or system calls [4]. A DISE-based reference monitor checks whether a program executing a certain instruction is permitted to do so. Three key properties make DISE attractive for enforcing security policies. First, the restricted PT/RT access model ensures that the security policies themselves are not tampered with. Second, DISE's positioning at the decoder and the atomic internal control-flow model of replacement sequences ensures that security checks implemented as DISE productions cannot be bypassed. Finally, DISE productions are small, posses private data registers, and are naturally expressed as declarative rules making them amenable to automated reasoning. DISE security policies combine the generality of software with the tamper and subversion resistance of hardware schemes.

3.2. Aware ACFs

In aware mode, DISE can be viewed as an interface for creating ACFs that combine static and dynamic components: the static component analyzes the program, defines the productions, and plants the codewords; the dynamic component expands the codewords and performs the actual "work."

Dynamic code decompression. Code size is an important concern for embedded systems. Static compression coupled with dynamic decompression address it. Dynamic decompressors come in two varieties. Those that sit on the instruction cache fill path [34] enable compressed memory images. Those that decompress fetched instructions [20] also enable a compressed cache footprint. High-performance processors may use decompressors of the second kind together with smaller instruction caches.

Using DISE, we implement post-fetch decompression as an aware ACF without ISA redesign and without decompression-specific hardware. Counter-intuitively, the DISE implementation enables more sophisticated compression than that supported by dedicated decompressors. DISE provides a mechanism (PT/RT reloading) for customizing the decompression dictionary to an application or even application phase. In addition, DISE enables the use of parameterized decompression templates that yield sequences with different register names or immediate values when instantiated with different arguments by different static codewords. This feature allows the compression of PC-relative branches. The problem with unparameterized compression of PC-relative branches is that compression itself changes relative PCs. Two static branches that can share a dictionary entry prior to compression (i.e., their PC-relative offsets were identical) may no longer be able to share it after compression is performed. Finding a stable dictionary configuration is difficult [20]. By making the PC-relative offset a replacement sequence parameter, DISE allows two static branches to share a dictionary entry. After compression, the offset of each static branch adjusted independently via then parameter. An example of parameterized DISE (de)compression (albeit without branches) is shown in Figure 4.

As an aware ACF, our implementation uses explicit replacement sequence tagging. Each decompression codeword consists of a reserved opcode (all use the same one), three 5-bit register or immediate parameters, and an 11-bit replacement sequence tag. A single pattern specification—which matches the reserved opcode—can name up to 2K dictionary entries. We use a greedy compression algorithm [20]. For each application, we build an exhaustive set of candidate dictionary entries: instruction sequences of any size that do not straddle basic blocks. We use

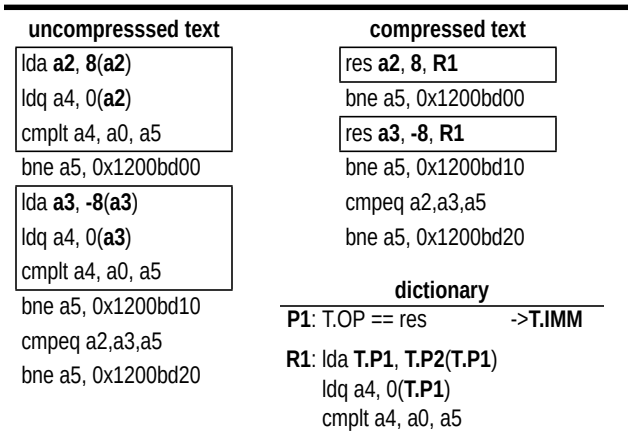

FIGURE 4. Dynamic code (de)compression

parameterization to combine candidate sequences. From this list, we iteratively choose the sequence that provides the greatest immediate compression. The compression calculation weighs the cost of coding the dictionary entry into the application's production segment, against the number of static instructions compressed out of the text segment. When the process terminates, we instantiate the decompression codewords with the appropriate parameters, including branch offsets.

Other aware ACFs. Several systems have been proposed for dynamically generating code that is specialized, i.e., customized to exploit particular runtime values [2, 13]. Unfortunately, even the most efficient software dynamic code generators have significant runtime overhead—from 10 to as many as 1000 cycles per generated instruction [13]—limiting their applicability and specialization granularity. DISE can act as a substrate for fast dynamic code generation. Consider this simple scenario: a loop containing a multiply instruction with one loop invariant operand. If the operand is a power of two, the multiplication can be reduced to a shift. In DISE, we implement this specialization by replacing the multiply with a codeword. At runtime, prior to entering the loop the value of the operand is tested and used to define the replacement instruction appropriately. This example appears trivial because a software specializer could easily rewrite the multiply instruction (assuming self-modifying code is permitted). The advantage of DISE is clearer when the invariant operand is the sum (or difference) of two powers of two allowing the multiply to be reduced to two shifts and an addition. With DISE, this specialization is just as easy as the first. A software specializer, however, would have to replace what was a single instruction with three, retarget branches around the expanded code, and scavenge a free register to hold the intermediate result.

While on the topic, we should note that DISE is not self-modifying code. DISE allows an application to modify its own code but in a structured and highly temporary way.

3.3. ACF Composition

DISE dynamically composes ACF and application code in a way that is transparent to the application. The use of logically separate storage and control facilitate this operation. Because it must be written specifically for this dynamic composition, ACF code is naturally declarative (functional), allowing ACFs to be dynamically composed with each other.

Composition semantics. The semantics of composition are defined by precedence or "nesting" relationships. DISE does not treat instructions in a replacement sequence as candidates for subsequent expansion. This restriction prevents infinite ACF recursion and simplifies the hardware. Nevertheless, when desired, this behavior can be simulated via composition. We say that ACF **X** is nested within ACF **Y** if the final instruction stream is equal to the one obtained by first applying ACF **X** to the application fetch stream—in functional notation **X**(application)—and then applying ACF **Y** to *that* stream—**Y**(**X**(application)). The productions for **X**-within-**Y** are simply the productions of **Y** *plus* the productions of **X** with the productions of **Y** "executed" on its replacement sequences. Non-nested composition is also possible, but counter-intuitively is more difficult. The difficulty lies in combining replacement sequences of identical or overlapping pattern specifications such that the original meanings of the component sequences are preserved. Designing algorithms for non-nested composition is an area for future work.

Composition is not a primitive exported by the DISE hardware system. DISE productions are composed in software. The precise mechanics of composition (i.e., which entity performs the composition and how) depends on the ACFs composed.

Transparent with transparent. Transparent ACFs are utilities that act on unmodified applications. Transparent ACFs are managed either by the OS kernel (e.g., fault isolation, reference monitoring) or by some other third-party application (e.g., code assertions). The managing entity—kernel or debugger—composes transparent ACFs in its own data space.

Examples of nested and non-nested transparent compositions—using productions for fault isolation and store address tracing—are shown in Figure 5. Store address tracing uses a single production that writes the store addresses to an array whose own address is in dedicated register $dr5. The composition at the bottom left of the figure nests address tracing within fault isolation—i.e., we fault isolate traced code. It consists of the fault isolation production, **R1**, and the fault isolation production applied to the address tracing replacement sequence, **R3**. **R3** contains two stores—the second one is the trigger, T.INSN—both of which are expanded by fault isolation's **P1**->**R1** production (in boxes). The first store in the address tracing replacement sequence is completely literal (i.e., it has no parameters), so the instantiation of the fault isolation replacement sequence within it (first box) uses

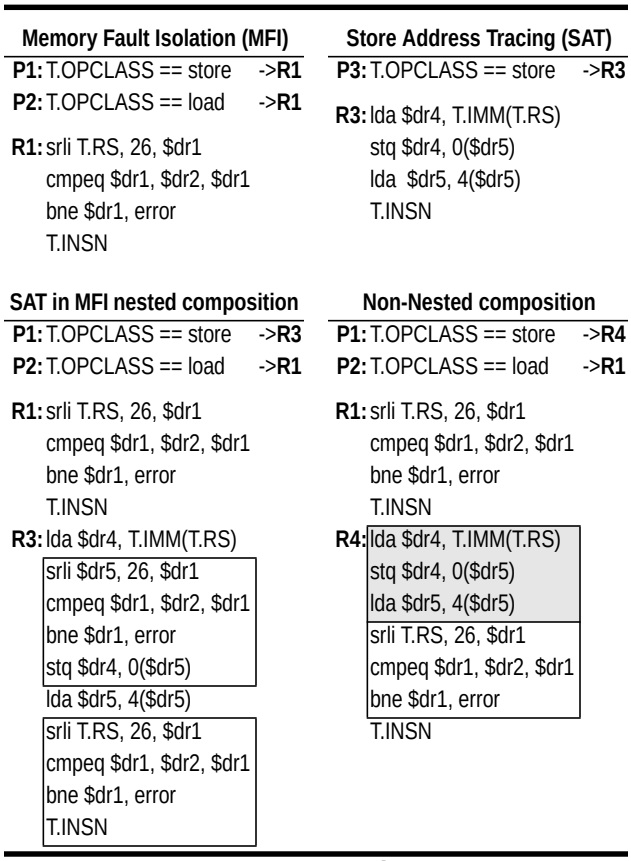

FIGURE 5. Composing ACFs

actual register names, replacing T.RS with $dr5. We call this ***replacement sequence inlining***. In general, inlining may require DISE registers to be renamed to avoid conflicts.

The bottom right hand side of the figure shows a non-nested composition. Here, we trace and fault isolate application stores, but do not fault isolate the stores that perform the tracing. The composition consists of the productions of both ACFs, where replacement sequences of overlapping pattern specifications (**P1** and **P3**) are merged in a way that preserves the meaning of both original sequences. Here, we merge the store productions, **R1** and **R3**, into a new production, **R4**. Here, a simple merging in which we concatenate the two replacement sequences—address tracing in the shaded box, fault isolation in the clear box—but leave only a single instance of the trigger, produces the desired result. For two arbitrary replacement sequences, this may not be the case (non-nested composition may in fact be impossible).

Aware with aware. Since aware ACFs require a static binary transformation (to insert the codewords), it is the responsibility of the binary rewriting utility to compose its own ACF with existing ones. Non-nested composition is relatively straightforward. The binary rewriter must only ensure that it does not include any of the codewords planted by other ACFs in its own replacement sequences (recall, recursive expansion is not allowed). It must also ensure that its own codewords do not collide with those used by any previously applied ACFs. An easy way to do this is to use a different reserved opcode. Alternatively, a single opcode could be used for two ACFs provided the replacement sequence identifiers used in explicit tagging do not collide. Nested composition is more involved. Here, the outer rewriter must not only transform the text, but also inline its own replacement sequences—using the procedure outlined above—into those of previously applied ACFs.

Transparent with aware. One of DISE's strengths is its composition of transparent and aware ACFs. Typical composition nests the transparent ACF within the aware one. For instance, when composing fault isolation and decompression, we want to fault isolate the uncompressed program, not the decompression codewords. Similarly, when composing code assertions with dynamic specialization, we want to apply the code assertions to the generated code, not to the codewords that generate it. The problem with this sort of composition in current environments is that aware ACFs are typically server-side customizations while transparent ones typically originate from the client. However, using conventional rewriting, the inner transparent ACF has to be applied first. As a concrete example, consider the fault isolation/decompression combination. To compose these using existing means we would first apply fault isolation to an application, then compress the result. The question is: why would a server store a compressed application that includes fault isolation code when fault isolation is a client option? By composing productions independently of the application, DISE allows the desired behavior to be implemented. The server compresses an unmodified application and delivers it to the client. The client applies the transparent fault isolation productions to the aware decompression replacement sequences via inlining.

Since the binary rewriter is not available to the client, and aware productions exist in the application's data segment rather than kernel memory, composition cannot be invoked by the kernel and performed in memory. Instead, we invoke composition from the RT miss handler on every aware ACF production miss, and represent composite productions in the RT only.

4. Experimental Evaluation

We evaluate DISE using simulation tools built on top of the SimpleScalar Alpha instruction set and system call definition modules [5]. The simulator models a MIPS R10000-like 4-way superscalar processor with a 12 stage pipeline, 128 entry reorder buffer, 80 reservation stations, and aggressive branch and load speculation. We model an on-chip memory hierarchy with 32KB instruction and data caches and a unified 1MB L2. The simulator models a DISE mechanism with a default configuration of 32 PT entries and 2K RT entries. Each PT and RT entry occupies 8 bytes (a highly conservative estimate), the total sizes of the two structures are 512 bytes and 16KB respectively. To model the DISE interface, we flush the pipeline on a PT or RT miss and stall for a fixed number of cycles: 30 for a simple miss, 150 for a miss requiring replacement sequence composition.

We apply ACFs to the SPEC2000 integer benchmarks. The benchmarks are compiled for the Alpha EV6 architecture with GCC 3.2.2 using the -O4 optimization flag. Results are reported using complete runs on *test* inputs. The simulator extracts nops from both the dynamic instruction stream and the static image.

4.1. Memory Fault Isolation

We compare DISE and binary rewriting implementations of memory fault isolation. Our metric is execution time normalized to the memory fault isolation-free case.

DISE formulation. The first and last bars in the top graph in Figure 6 compare the binary rewriting implementation with two DISE schemes. *DISE4* uses the same four-instruction replacement sequences used by binary rewriting. *DISE3* exploits DISE's semantics that disallow jumps into the middle of replacement sequences to eliminate a copy instruction. In both experiments, we simulate *free* (with no additional runtime cost) DISE. DISE4 outperforms binary rewriting: while they retire an identical number of instructions, DISE4 does not place the ACF instructions into the cache, and incurs fewer misses. DISE3 outperforms both by also executing fewer instructions.

DISE implementation. A free implementation of DISE may not be possible. Fitting DISE into the decoder may require either adding another decoding stage or incurring a single cycle stall on every successful DISE expansion. The performance of these two options are shown as the middle two bars—+*stall* and +*pipe*, respectively—in the top graph of Figure 6. The effects are intuitive: the penalty of an additional decoding stage is proportional to the frequency of mispredicted branches, typically around 1%. The penalty of a single-cycle stall per expansion is proportional to the total number of expansions. The stall option fits better with our design philosophy of zero performance degradation on ACF-free code. Unfortunately, expansion frequency is often much higher than branch misprediction frequency. Fault isolation, for instance, expands about 30% of dynamic instructions. For programs that exhibit good instruction cache behavior even after rewriting (e.g., *bzip2*, *gzip*, *mcf*, *vpr*), the stalling DISE implementation underperforms rewriting. The serial stall also dominates the cost/saving of additional instructions, narrowing the performance gap between

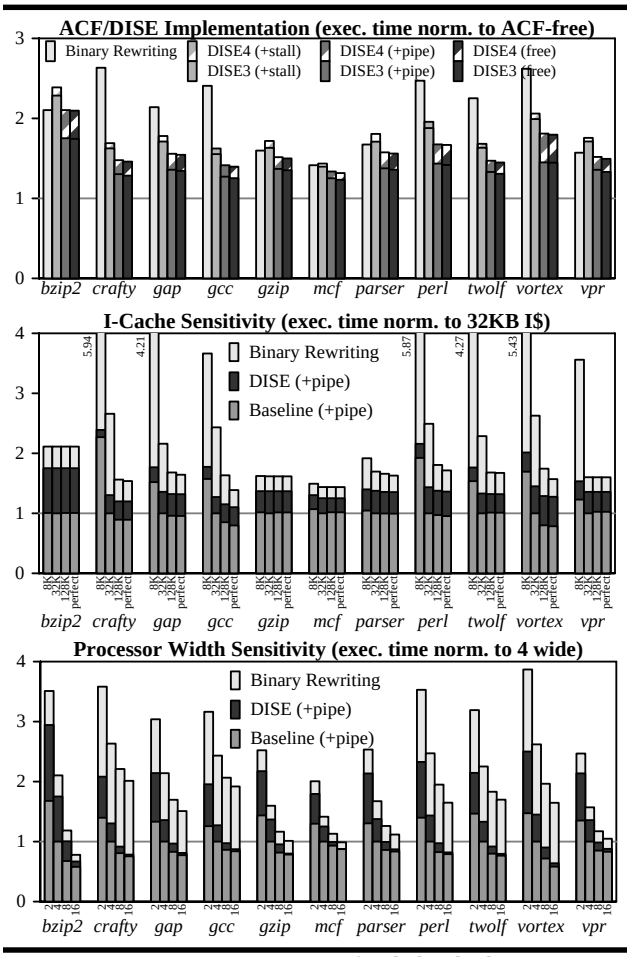

FIGURE 6. Memory fault isolation

DISE4 and DISE3. We are left to choose between a uniform 1% penalty that applies even to ACF-free codes and a substantial penalty when ACFs that result in frequent replacements are used. If heavy DISE use is projected, the elongated pipeline may be the sensible choice. For the remainder of the evaluation, we assume this design.

Cache size and processor width. ACF code has two costs. The static cost is decreased effective instruction cache capacity. The dynamic cost is decreased effective pipeline throughput. In general, DISE ACFs have only the dynamic cost. As in memory fault isolation, this cost is often lower because DISE's atomic replacement sequence semantics enable simpler, shorter formulations. The bottom graphs of Figure 6 isolate these costs, by relaxing cache capacity and pipeline bandwidth constraints, respectively. The middle graph shows relative execution times of DISE3 and binary rewriting on 4-wide processors with instruction caches of varying sizes. As cache size increases, static overhead decreases and the dynamic overhead remains constant. For the binary rewriting implementation, 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. These trends favor DISE. Physical cache size is limited by access latency while instruction working sets are growing.

The bottom graph shows relative performance on 32KB instruction-cache processors of different widths. At high widths, data dependences limit parallelism within a fixed reordering window, allowing ACF code to exploit idle resources at little perceived cost. Although this trend is apparent for DISE, the binary rewriting implementation does not improve as rapidly with wider machines. While increased processor width reduces the dynamic cost of ACFs, the static cost remains and, in fact, becomes relatively larger. As the absolute cost of the application shrinks, the relative cost of each cache miss grows. This trend also bodes well for DISE: its advantage over binary rewriting will increase as processor performance grows.

4.2. Dynamic Code Decompression

DISE can be used to implement post-fetch code decompression, enabling both reduced static code sizes and improved performance due to better instruction cache utilization. We compare the DISE implementation with a dedicated decoder-based decompressor [20]. We assume that the dedicated decompressor has a facility for programming the on-chip dictionary. Our metric is static code size, normalized to uncompressed text size.

DISE features. Several features distinguish DISE decompression from its dedicated counterpart [20]. In DISE's favor, it supports parameterized decompression, which enables more efficient dictionary usage and the compression of PC-relative branches. Parameterization does, however, increase the size of dictionary entries. On the other hand, the dedicated decompressor may use 2-byte codewords which improve compression in two ways: reducing the size of compressed representations, and making the compression of single instructions profitable. DISE *can* exploit existing short formats, but we do not account for that possibility here.

The effects of these features are separated in the top graph of Figure 7. Stacked bars show the results of six experiments. The bottom portion of each stack is compressed code size, the top portion (solid black) adds the dictionary size. The first experiment (*dedicated*) is the dedicated decompressor, complete with 2-byte codewords and single-instruction compression. The compression ratios achieved—about 75% of original text size (note the scale of the graph), dictionary not included—are comparable to those previously published [20]. In the next two bars, we progressively eliminate the dedicated decompressor's two advantages: single-instruction compression (*–1insn*), and the use of 2-byte codewords (*–2byteCW*). Eliminating these features, reduces compression effectiveness to 85%. With dedicated-decompression-specific features removed, the next three bars add DISE-specific features. The use of parameterization requires four additional bytes per dictionary entry to hold the instantiation directives (*+8byteDE*). Without parameterization, larger dictionary entries require more static instances to be considered profitable. As a result, fewer of them are selected and compression ratios degrade to 90% and above. Shown in the fifth bar, parameterization (*+3param*, we allow three parameters per dictionary entry) more than compensates for the increased cost of each dictionary entry by allowing sequences with slight differences to share entries; it improves compression ratios dramatically (back

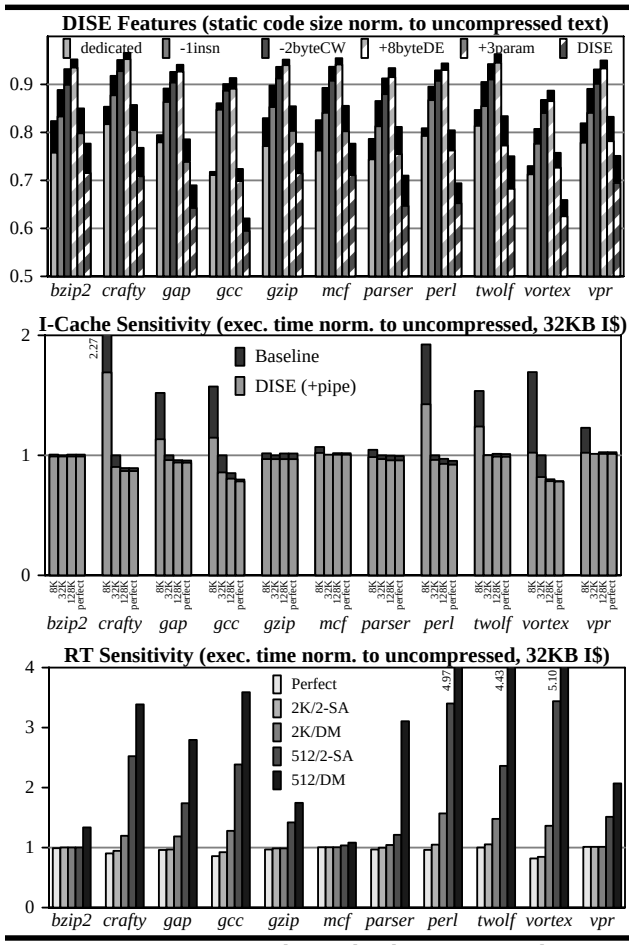

FIGURE 7. Dynamic code decompression

down to 75%). The final bar (*DISE*)—corresponding to the full-featured DISE implementation—adds the compression of PC-relative branches. The high static frequency of PC-relative branches enables compression ratios of 65%, appreciably better than those achieved with dedicated hardware.

Performance. Compression can improve performance by reducing instruction cache misses. The middle graph in Figure 7 shows execution times for DISE decompression and four instruction cache sizes: 8KB, 32KB, 128KB and perfect (infinite). Times are normalized to the uncompressed, 32KB cache case. Most of the SPEC2000 benchmarks—except for *crafty*, *gzip*, and *vpr*—have uncompressed instruction working sets smaller than 32KB. About half have working sets larger than 8KB and suffer significant degradation at that cache size. The addition of a 2K entry RT (we model a perfect RT in this experiment, but the results are equivalent as discussed in the next paragraph)—16KB of on-chip decompression dictionary storage—can compensate for a significant fraction of this loss. Of course, an RT can be used for purposes other than compression. Increasing cache size may increase fetch latency and does not allow the program image to be compressed in other levels of the memory hierarchy.

RT size. For the results reported above, we modeled a perfect RT. In the bottom graph of Figure 7, we evaluate the impact of realistic RTs using four configurations, 512 and 2K entries, each both direct-mapped and 2-way set-associative. An RT miss prompts a pipeline flush and 30 cycle stall. For our dictionaries —which were selected to minimize combined program and dictionary size without regard to RT misses—a 2K, 2-way RT (nearly) matches the perfect RT in all benchmarks. The direct-mapped configuration performs nearly as well. The effectiveness of 512-entry RTs depends on production working set size (which is smaller than the instruction working set size). While the performance of benchmarks with small production working sets (e.g., *bzip2*, *mcf*, *parser*) remains good, especially with the set-associative RT, that of the larger benchmarks degrades significantly. Naturally, the trade-off between performance (RT hit rate) and compression ratio must be made appropriately.

4.3. Composing Decompression and Fault Isolation

One of the benefits of DISE is that it supports dynamic ACF composition. Here, we measure the performance of simultaneous decompression/fault isolation. Thus far we have seen two implementations of each: binary rewriting and DISE for fault isolation, dedicated hardware and DISE for decompression. These yield three combinations for implementing their composition: binary rewriting for fault isolation and dedicated decompression, binary rewriting for fault isolation and DISE decompression, and DISE for both. The fourth combination—DISE for binary rewriting and dedicated hardware decompression—is possible, but because DISE is so physically similar to a dedicated decompressor, a single processor is unlikely to include both mechanisms. As previously noted, the solutions that use binary rewriting for fault isolation do not fit real world code usage models. Here, we focus solely on performance.

Performance. The top graph of Figure 8 shows the performance of the three ACF combinations on processors with four different cache sizes, normalized to an unmodified execution on a

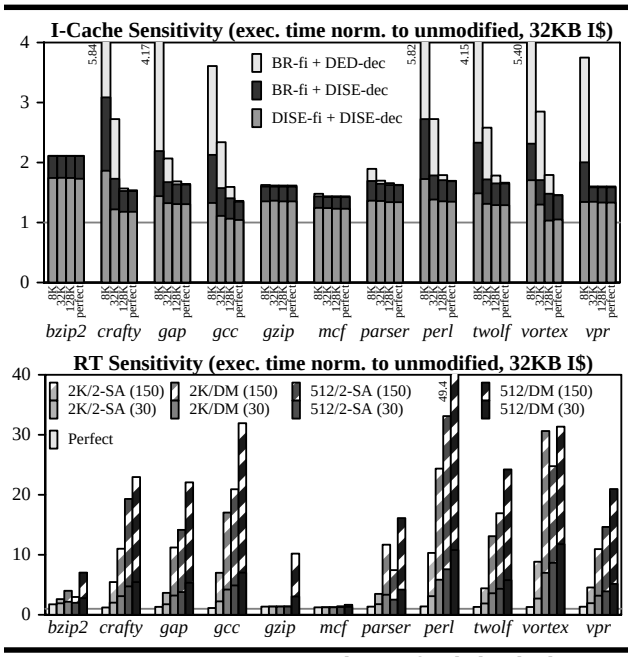

FIGURE 8. Decompression + fault isolation

processor with a 32KB cache. Again, we use a perfect RT. The rewriting/dedicated (fault-isolation/decompression) combination performs poorly, especially for small caches. The rewriting implementation of fault isolation bloats the text to such a degree that dedicated compression cannot compensate. Factors of 2–3 performance degradation are observed for a 32KB cache, 4 and higher for an 8KB cache. Combining DISE (de)compression with binary rewriting helps considerably. DISE's parameterized compression and ability to compress PC-relative branches allow it to essentially reverse most of the code bloat caused by rewriting, the fault isolation sequences are simply factored out into decompression productions. The cost of the registers lost to scavenging by the rewriter remains, as does the inefficiency of compressing a bloated executable, no matter how redundant. These costs disappear in the DISE-DISE implementation, which only incurs the dynamic cost of the fault isolation instructions and any costs associated with RT performance. We examine these next.

RT size and miss latency. Composing ACFs degrades RT performance in two ways: effective capacity is reduced by inlining replacement sequences into one another, and the composition operation increases RT miss handler latency. These effects are quantified and isolated in the second graph of Figure 8. The bottom portion in each stack measures performance degradation due to RT capacity reduction. The 2K, set-associative RT continues to perform well, producing slowdowns of 0–80%, much lower than the factors of 2, 3, and above slowdowns produced by non-DISE solutions (see section above). The 512 and direct-mapped configurations suffer due to the greatly increased RT working set, yielding factor of 5 and above slowdowns. For the top portion (striped), we increase RT miss handler latency from 30 to 150 cycles to model the latency of composition. Here, performance degradations of factors of up to 50 are observed, although for a 2K, 2-way RT configuration, factors of 5 slowdowns are the norm. Certainly, the performance of composed ACFs can be improved by accounting for RT size *a priori* and making the appropriate usage trade-offs. However, improving the performance characteristics of this trade-off for composed ACFs is an important area of future work.

5. Related Work

DISE builds on several large bodies of work.

Hardware translation/expansion. IA32 processors [12, 14, 15] dynamically macro-expand each CISC instruction into one or more internal RISC operations, potentially caching the translations [12]. Dynamic Instruction Formatting [22] schedules cached instructions into VLIW groups. Speculative Decode [18] implements microarchitectural execution time optimizations like silent-store elimination using alternate expansions. These facilities resemble DISE mechanically, but differ in two major ways. First, they translate the ISA to a simpler form for the purpose of reducing execution complexity. DISE adds ISA instructions in order to add functionality. Second, they are inaccessible to software, and thus capable only of changing/optimizing representations. To add functionality, DISE has an API.

Hardware ACF implementations. Programmable microcode [6, 23] was an early choice for implementing ACFs like address tracing [1]. Although microcode remains a viable option for implementing complex instructions and programmable microcode stores persist (e.g., Intel supports limited microcode patching to fix bugs in the field [16]), its current use is too sparse and irregular to effectively support ACFs. DISE implements ACFs using ISA instructions. Alpha's PAL [29] exposes hardware internals to privileged software, but is invoked using calls and traps, not matching and replacement, making it unsuitable for implementing the ACFs we describe here.

The profiling processor [35] and instruction path co-processor [7] provide additional functionality—profiling and trace construction, respectively—at virtually no cost to the application using dedicated, potentially programmable, pipeline stages. To minimize performance impact, the dedicated stages are placed post retirement and thus are inappropriate for ACFs which must inspect or modify instructions before they execute. By transforming the instruction stream before execution, DISE supports a different (potentially broader) class of ACFs. In decoder-based decompression [20], tagged instructions in a compressed application are interpreted as dictionary indices and replaced by the corresponding entries. DISE generalizes this functionality, using parameterized matching and replacement, and thus can be used for other purposes. DISE is not reconfigurable hardware**,** it reconfigures the instruction stream, not the functional units.

Software translation/expansion. FX!32 [31], DAISY [11] and Transmeta's Crusoe [17] convert one ISA to another in software. Dynamo [2] adds optimization. A software translator may implement certain ACFs itself (e.g., profiling) or have the ability to add simple ACFs to translated code. However, using a translator for general purpose ACF implementation is difficult and inconvenient, especially if translation itself is not needed (i.e., the application is native). DISE can be used to add ACF code to translated code. DISE is not suitable for emulation, as it only recognizes native instructions.

Software ACF implementations. Binary rewriting tools like Atom [30], Etch [25], and EEL [19] provide hooks for adding ACF code to an application. Paradyn [21] includes a binary rewriter that can transform any program, including the OS kernel, while it runs. Binary rewriters have been used to implement profiling [3], dynamic race detection [27], shared memory communication [28] and memory fault isolation [32]. These can all be implemented using DISE without degrading cache performance or incurring the overhead of rewriting the executable.

ACF interfaces. DELI [9] is an API to Dynamo's caching, linking, and optimization infrastructure. Like DISE, DELI provides an interface to an execution substrate to help with the implementation of ACFs. Implemented in software, DELI is more heavyweight than DISE, but can also be used for tasks like emulation. DISE can be used both with DELI or by it.

6. Conclusion

There is a growing number of computing platforms (e.g., server, workstation, palmtop, phone, etc.), and each has its own unique set of requirements. The ability to customize an application to fit the requirements of a particular environment is becoming an important system function. ***Dynamic instruction stream editing (DISE)*** is a cooperative software-hardware mechanism that dynamically customizes applications via programmable

instruction macro-expansion. DISE executes a set of productions on an application's fetch stream, feeding the execution engine an enhanced stream that includes application customization functions (ACFs). DISE is a single facility that supports two general models of ACFs: transparent and aware. We have demonstrated DISE formulations of both. Memory fault isolation is a transparent ACF implemented via productions that expand "naturally occurring" instructions. Dynamic code decompression is an aware ACF implemented via productions that re-expand specially crafted codewords back to their original instruction sequences. We showed how DISE can dynamically add fault isolation code to an application as it is decompressed to illustrate how the DISE facility can dynamically compose multiple ACFs.

DISE combines the advantages of software and hardware ACF implementations. Like software approaches, DISE is expressive, programmable, and uses the underlying hardware to dynamically distribute execution resources between application and ACF code. Like hardware approaches, DISE need not pay the cost of transforming a program binary. DISE's hardware components require modest, localized changes to the decoding pipeline and in some cases can use existing hardware structures.

Our experiments show that DISE has (often significantly) better performance than software-based ACF implementations, and architectural trends suggest DISE's advantages will grow with time. In addition, DISE performance is competitive with that of hardware-only ACF implementations. Of course, DISE's clear advantage over these is its general-purpose nature.

There are several avenues for future work. DISE currently uses a small, expressive set of features—parameterized matching and replacement, dedicated register storage—to support many ACFs. As our experience with DISE grows and more ACFs are implemented, this set may be generalized or expanded. For instance, the incorporation of runtime data values as replacement instruction constants has applications in dynamic code optimization. Streamlining the hardware implementation, refining the interface, and better integration with the OS are important as well. A final area is the construction of tools for managing DISE-style ACF code, including routines for safety-analysis and composition, and even ACF-specific languages.

Acknowledgments

We thank the anonymous referees for their suggestions. Amir Roth is supported by NSF award CCR-0238203.

References

[1] A. Agarwal, R. Sites, and M. Horowitz. "ATUM: A New Technique for Capturing Address Traces Using Microcode." In *ISCA-13*, May 1986.

[2] V. Bala, E. Duesterwald, and S. Banerjia. "Dyanmo: A transparent dynamic optimization system." In *PLDI-2000*, Jun. 2000.

[3] T. Ball and J. Larus. "Optimally Profiling and Tracing Programs." In *POPL-19*, 1992.

[4] M. Bernaschi, E. Gabrielli, and L. Mancini. "Operating System Enhancements to Prevent the Misuse of System Calls." In *CCS-7*, 2000.

[5] D. Burger and T. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[6] R. E. Calcagni and W. Sherwood. "Patchable Control Store for Reduced Microcode Risk in a VLSI VAX Microcomputer." In *17th Microprogramming Workshop*, 1984.

[7] Y. Chou, J. Fung, and J. Shen. "Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection." In *ICS-13*, Jun. 1999.

[8] M. Corliss, E. Lewis, and A. Roth. "DISE: Dynamic Instruction Stream Editing." Technical Report MS-CIS-O2-24, University of Pennsylvania, Jul. 2002.

[9] G. Desoli, N. Mateev, E. Deusterwald, P. Faraboschi, and J. Fisher. "DELI: A New Run-Time Control Point." In *MICRO-35*, Nov. 2002.

[10] K. Diefendorf. "K7 Challenges Intel." *Microprocessor Report*, 12(14), Nov. 1998.

[11] K. Ebcioglu and E. Altman. "DAISY: Dynamic Compilation for 100% Architectural Compatibility." In *ISCA-24*, Jun. 1997.

[12] P. Glaskowsky. "Pentium 4 (Partially) Previewed." *Microprocessor Report*, 14(8), Aug. 2000.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. "Annotation-Directed Run-Time Specialization in C." In *PEPM-97*, Jun. 1997.

[14] L. Gwenapp. "Intel's P6 Uses Decoupled Superscalar Design." *Microprocessor Report*, 9(2), Feb. 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] T. Halfhill. "Transmeta Breaks x86 Low-Power Barrier." *Microprocessor Report*, Feb. 2000.

[18] I. Kim and M. Lipasti. "Implementing Optimizations at Decode Time." In *ISCA-29*, May 2002.

[19] J. R. Larus and E. Schnarr. "EEL: Machine-Independent Executable Editing." In *PLDI-95*, June 1995.

[20] C. Lefurgy, P. Bird, I.-C. Cheng, and T. Mudge. "Improving Code Density Using Compression Techniques." In *MICRO-30*, Dec. 1997.

[21] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn Parallel Performance Measurement Tools." *IEEE Computer*, 28(11), 1995.

[22] R. Nair and M. Hopkins. "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups." In *ISCA-24*, Jun. 1997.

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

[24] T. Reps. "The Use of Program Profiling in Software Testing." In *Informatik'97*. Springer-Verlag, Sep. 1997.

[25] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. "Instrumentation and Optimization of Win32/Intel Executables Using Etch." In *USENIX Windows NT Workshop*, August 1997.

[26] T. Rooker. "The Reference Monitor: An Idea Whose Time Has Come." In *1993 Workshop on New Security Paradigms*, 1993.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. "Eraser: A Dynamic Race Detector for Multi-Threaded Programs." *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.

[28] D. Scales, K. Gharachorloo, and C. Thekkath. "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory." In *ASPLOS-7*, Oct. 1996.

[29] R. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[30] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools." In *PLDI-94*, Jun. 1994.

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

[32] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. "Efficient software-based fault isolation." In *SOSP-14*, Dec. 1993.

[33] D. Wallach, D. Balfanz, D. Dean, and E. Felten. "Extensible Security Architectures for Java." In *SOSP-16*, 1997.

[34] A. Wolfe and A. Chanin. "Executing compressed programs on an embedded RISC architecture." In *MICRO-25*, 1992.

[35] C. Zilles and G. Sohi. "A Programmable Co-processor for Profiling." In *HPCA-7*, Jan. 2001.