A DISE Implementation of Dynamic Code Decompression

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

ABSTRACT

Code compression coupled with dynamic decompression is an important technique for both embedded and general-purpose microprocessors. *Post-fetch decompression*, in which decompression is performed after the compressed instructions have been fetched, allows the instruction cache to store compressed code but requires a highly efficient decompression implementation. We propose implementing post-fetch decompression using *dynamic instruction stream editing* (DISE), a programmable decoder—similar in structure to those in many IA32 processors—that is used to add functionality to an application by injecting custom code snippets into its fetched instruction stream. A DISE implementation of post-fetch decompression naturally supports customized program-specific decompression dictionaries, enables parameterized decompression allowing similar instruction sequences to share dictionary entries, and uses no decompression-specific hardware. Cycle-level simulation of DISE decompression shows that it can reduce static program size by 35% and execution time by 20%. Parameterized decompression, a feature unique to DISE, accounts for 20% of the code size reduction by making more effective use of the dictionary and allowing PC-relative branches to be included in compressed sequences. DISE-based compression can reduce total energy consumption by 10% and the energy-delay product by as much as 20%.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; C.1 [**Computer Systems Organization**]: Processor Architectures

General Terms

Performance, Design, Experimentation

Keywords

Code compression, code decompression, DISE

1. INTRODUCTION

Code compression coupled with dynamic decompression is a useful technique in many computing contexts. Certainly, (de)compression benefits embedded devices where power, size, and cost constraints force the use of small caches and memories. But general-purpose systems can benefit from the technique as well. Not only is power a growing concern for these systems, they can also use (de)compression to improve performance.

Dynamic code decompression techniques are characterized by *when* they perform decompression. Several systems integrate decompression into the instruction cache fill path [12, 22]. The advantages of fill-path decompression is that it allows the use of unmodified cores while incurring the decompression penalty on instruction cache misses only. Its disadvantages are that it stores uncompressed code in the instruction cache and requires a mechanism for translating instruction addresses from the uncompressed image (in the pipeline and instruction cache) to the compressed one (in memory). An alternative approach decompresses instructions after they are fetched from the cache but before they enter the execution engine [16]. *Post-fetch decompression* requires a modified processor core and an ultra-efficient decompression implementation, because every fetched instruction must at the very least be inspected for possible decompression. However, it allows the instruction cache to store code in compressed form and eliminates the need for a compressed-to-decompressed address translation mechanism (only a single static version of the code exists, the compressed one).

In this paper, we propose an implementation of post-fetch code decompression via *dynamic instruction stream editing* (DISE) [7]. DISE is a hardware-based instruction macro-expansion facility similar in structure and function to IA32 CISC-instruction-to-RISC-microinstruction decoders. However, it is both programmable and not specific to CISC ISAs. Rather than merely changing the representation of the fetched instruction stream, DISE uses the expansion process to augment or modify its *functionality* by splicing custom code snippets into it. DISE is a single mechanism that unifies the implementation of a large number of functions (*e.g.*, memory fault isolation, profiling, assertion checking, *etc.*) that, to date, have been implemented in isolation using *ad hoc* structures. The hardware components of DISE (less the programming interface) are well understood and already exist in many IA32 microprocessors [9, 10, 11].

A DISE implementation of dictionary-based post-fetch decompression has several important virtues. DISE's macro-expansion functionality enables parameterized (de)compression, an extension to conventional decompression that allows multiple, similar-but-not-identical decompression sequences to share dictionary entries, improving dictionary space utilization and allowing PC-relative branches to be included in compressed instruction sequences. DISE's programming interface also allows the decompression dictionary to be customized on a per application basis, further improving compression. Finally, as a general purpose mechanism, DISE

LCTES'03, June 11–13, 2003, San Diego, California, USA.

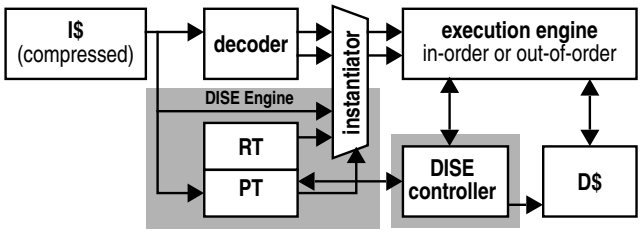

Figure 1: DISE structures in processor pipeline.

can implement many other features and even combine them (dynamically) with decompression.

We evaluate DISE decompression via cycle-level simulation. On the SPEC2000 and MediaBench benchmarks, the DISE (de)compression implementation enables code size reductions of over 35% and performance improvements (execution time reductions) of 5–20%. Parameterized decompression, a feature unique to the DISE implementation, accounts for 20% (absolute measure). We also show that dictionary programmability is an important consideration for dynamic decompression. Although previous post-fetch decompression proposals do not preclude programmability and may even assume it, none evaluates its importance or provides a mechanism for its implementation. Finally, we show that DISE-based compression can reduce total energy consumption by 10% and the energy-delay product by as much as 20%.

The remainder of the paper is organized as follows. The next section introduces DISE. Section 3 presents and discusses our DISE implementation of dynamic code (de)compression, and Section 4 evaluates it. The final two sections summarize related work and conclude.

2. DISE

Dynamic instruction stream editing (DISE) [7] is a facility for implementing *application customization functions* (ACFs). ACFs customize a given application for a particular execution environment. Examples of ACFs include profiling, dynamic optimization, safety checking, and (de)compression. Traditional ACF implementations have been either software or hardware only. The software solutions inject instructions into the application's instruction stream, but require expensive binary modification. The hardware approaches customize the application using dedicated pipeline stages, but are functionally rigid. DISE is a cooperative software-hardware mechanism for implementing ACF. Like software, DISE adds/customizes application functionality by enhancing/modifying its execution stream. Like hardware, DISE transforms the dynamic instruction stream, not the static executable.

DISE inspects every fetched instruction and macro-expands those matching certain patterns into parameterized instruction sequences. The expansion rules that encode instruction patterns and replacement sequences—called *productions*—are software-specified. From a hardware standpoint, DISE is similar to the CISC-instruction-to-RISC-microinstruction decoders/expanders used in IA32 microprocessors [9, 10, 11]. To these, DISE adds a programming interface that allows applications to supplement their own functionality and trusted entities (*i.e.*, the OS kernel) to augment or modify the functionality of other applications. Currently, decoder-based macro-expansion is used to simplify the execution engines of CISC processors. DISE performs a different function (adding functionality to an executing program) and so may be used in RISC processors as well. In this section, we describe those DISE features that are salient to decompression.

As shown in Figure 1, the DISE hardware complex is comprised of two blocks (shaded). The *DISE engine* performs matching and expansion of an application's fetch stream (described below). The DISE engine is a part of the processor's decode stage and its components—the *pattern table* (PT), *replacement table* (RT) and *instantiation logic*—are quite similar to corresponding structures—match registers, microinstruction ROM, and alias mechanism, respectively—already present in IA32 microprocessors to convert CISC ISA instructions to internal RISC microinstruction sequences [9, 10, 11]. The *DISE controller* provides an interface for programming the PT and RT, abstracting the microarchitectural formats of patterns and replacement instructions from DISE clients. While the PT and RT are continuously active (unless disabled), the DISE controller is a co-processor that is only activated when the DISE engine is configured (*i.e.*, rarely).

DISE engine. The DISE engine—PT, RT, and instantiation logic—matches and potentially replaces/expands every instruction in an application's fetch stream. The PT contains instruction patterns specifications. These patterns may include any part of the instruction itself: opcode, logical register names, or immediate field. Thus, for example, DISE is able to match instructions of the form, "stores that use the stack pointer as their base address." The RT houses specifications for the instruction sequences (called *replacement sequences*) that are spliced into the instruction stream when there is a match in the PT. A PT match produces an *RT identifier*, naming the RT-housed replacement sequence associated with the matching instruction pattern. A given PT entry can store an RT identifier directly, or indicate which bits in the matching instruction are to be interpreted as the identifier. The reason for this dual mode of operation is discussed below.

To enable interesting functionality, replacement sequences are parameterized. An RT entry—corresponding to a single replacement instruction—contains a replacement literal and a series of instantiation directives that specify how the replacement literal is to be combined with information from the matching instruction to form the actual replacement instruction that will be spliced into the application's execution stream. The instantiation logic executes the 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." The PT and RT entries for this particular production are shown (logically) in Figure 2. The PT entry matches the opcode of the instruction only. The two-instruction replacement sequence makes heavy use of parameterization (*e.g.*, for the second replacement instruction we simply copy every field from the original fetched instruction).

In addition to matching and parameterized replacement, DISE has several features—notably the use of a dedicated register space, and replacement sequence internal control—that simplify ACF implementation and improve ACF performance. Neither of these features is used in (de)compression.

DISE usage modes. DISE has two primary usage modes. In *application-transparent* mode, it operates on unmodified executables using productions that match "naturally-occurring" instructions with conventional opcodes. Examples of transparent ACFs include branch and path profiling (productions are defined for control transfer instructions) and memory fault isolation (productions are defined for loads and stores). In *application-aware* mode, DISE uses productions for *codewords*, specially-crafted instructions that do not occur naturally which are planted in the application by a DISE-aware rewriting tool. Codewords are typically constructed using reserved opcodes. Code decompression is an example of aware functionality. A DISE-aware utility compresses the origi-

PT

OP	R1	R2	RD	IMM	RTID
ldq	–	–	–	–	0

RT

ID	LITERAL	DOP	DR1	DR2	DRD	DIMM
0	andi –, 00ff, –	–	R2	–	R2	–
0	– –, –(–)	OP	R1	R2	RD	IMM

Figure 2: Sample DISE PT/RT entries.

nal executable by replacing common multi-instruction sequences with decompression codewords. At runtime, DISE replaces these codewords with the appropriate original instruction sequences.

The two usage modes correspond to the two methods of specifying RT identifiers. Transparent productions match "naturally-occurring" instructions whose raw bits cannot be interpreted as RT identifiers. For these, RT identifiers are stored directly in the PT entries (as in Figure 2). Aware productions must map a small number of reserved opcodes (perhaps even just one) to a large number of replacement sequences, and thus store RT identifiers in the planted DISE codewords.

DISE interface. DISE access is mediated by two layers of abstraction and virtualization. The DISE controller abstracts the internal formats of the PT and RT allowing productions to be specified in a language that resembles an annotated version of the processor's native ISA. The controller also virtualizes the sizes of the PT and RT, using a dedicated fixed-size counter table and RT tagging to detect and signal PT and RT misses, respectively. RT and, to a lesser degree, RT virtualization are crucial to improving the utility, generality, and portability of DISE ACFs. The OS kernel virtualizes the set of active productions to both preserve the transparency of multiprogramming and secure processes from malicious productions defined by other applications. OS kernel mediation does allow applications direct control over DISE productions that act on their own code.

The PT and RT are the top, "active" components of the DISE production memory hierarchy. DISE productions are encoded into executables in a special *.dise* segment, and are paged into main memory via traditional virtual memory hardware (they may also be created in memory directly). From memory, the productions may enter the processor either through the instruction memory structures or the data memory ones. The instruction path is attractive because it passes through the conventional decoder. The data path is preferable because productions often need to be manipulated (more on this shortly). A good compromise is to treat productions as data elements, but provide a DISE-controller managed path for passing them through the decode stage en route to the RT. The mechanics of moving productions from memory to the PT and RT (either imperatively or on a miss) resemble those of handling software TLB misses (*n.b.*, not page fault)—the thread is serialized by a short handler—and have similar costs.

The primary use of production manipulation, aside from the dynamic creation of productions, is the composition of multiple ACFs. DISE is a general facility that can implement a wide range of transparent and aware ACFs, both in isolation and *together*. Composition is performed by merging the productions sets of ACFs and applying the productions of one to the replacement sequences of the other. Previous work showed how decompression could be composed with memory fault isolation, a security ACF that inspects/isolates an application's memory operations [7]. Composition is a unique and powerful DISE feature that enables new software usage models. Although the composition of decompression with other ACFs is interesting, we do not evaluate it further here.

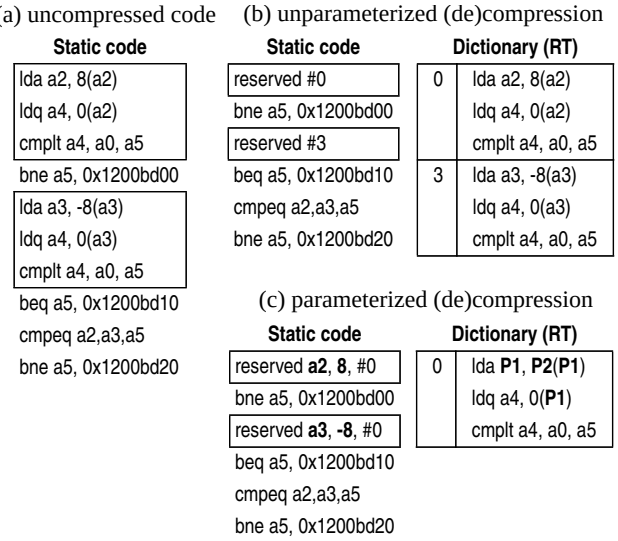

Figure 3: (De)compression examples.

3. (DE)COMPRESSION WITH DISE

DISE enables an implementation of dictionary-based post-fetch decompression that is functionally similar to a previously described scheme [16]. The DISE implementation is unique in that it supports parameterized decompression, has a programming interface that allows program-specific dictionaries, and uses hardware that is not decompression-specific. We elaborate on how DISE may be used to perform dynamic decompression and present our compression algorithm.

3.1 Dynamic Decompression

A DISE decompression implementation uses the RT to store the decompression dictionary. Decompression is an "aware" ACF. A DISE-aware compressor replaces frequently occurring instruction sequences with DISE codewords, which are recognized by their use of a single reserved opcode. DISE decompression uses a single PT entry to match all decompression codewords via the reserved opcode, and the codeword itself encodes the RT identifier of the appropriate replacement sequence. This arrangement is basically the same as the one used by the previously described scheme. However, to support parameterized decompression, DISE also uses some non-opcode bits of a codeword to encode register/immediate parameters. The parameter/RT-identifier division is flexible and may be changed on a per-application basis. For instance, in a 32-bit ISA with 6-bit opcodes, we could use 2K decompression entries (11 identifier bits) and up to 3 register/immediate parameters of 5 bits each (15 bits total). Alternatively, the 26 non-opcode bits could be divided to allow the specification of up to 64K decompression entries (16 bits) with each using up to 2 parameters (10 bits).

Parameterized (de)compression. Register/immediate parameters encoded into decompression codewords exploit DISE's parameterized replacement mechanism to allow more sophisticated compression than that supported by dedicated (*i.e.*, dictionary-index only) decompressors. In DISE, a single decompressed code template may yield decompressed sequences with different register names or immediate values when instantiated with different "arguments" from different static locations in the compressed code. In this way, parameterization can be used to make more efficient use of dictionary space. The use and benefit of parameterized decompression is illustrated in Figure 3. Part (a) shows uncompressed

static code; the two boxed three-instruction sequences are candidates for compression. Part (b) shows the static code and the dictionary (RT) contents for unparameterized compression. Since the sequences differ slightly, they require separate dictionary entries. With parameterized decompression, part (c), the two sequences can share a single parameterized dictionary entry. The entry uses two parameters (shown in bold): **P1** parameterizes the first instruction's input and output registers and the second instruction's input register, **P2** parameterizes the first instruction's immediate operand. To recover the original uncompressed sequences, the first codeword uses **a2** and **8** as values for the two parameters, while the second uses **a3** and **-8**, respectively.

In addition to allowing more concise dictionaries, parameterization permits the compression of sequences containing PC-relative branches. Conventional mechanisms are incapable of this because compression itself changes PC offsets. Although two static branches may use the same offset before compression, it is likely this will not be true after compression. General solution of this conflict is NP-complete [20]. In fact, post-compression PC-relative offset changes are no longer a problem. Multiple static branches that share the same dictionary entry prior to compression can continue to do so afterward. With parameterization, even branches that use different *a priori* offsets can share a dictionary entry. The one restriction to incorporating PC-relative branches into (de)compression entries is that their offsets must fit within the width of a single parameter. This restriction guarantees that no iterative rewriting will be needed, because compression can only reduce PC-relative offsets. As we show in Section 4, the ability to compress PC-relative branches gives a significant benefit, because they represent as much as 20% of all instructions.

Parameterization is effective for two reasons. First, only a few parameters are needed to capture differences between similar sequences. This is due to the local nature of register communication of common programming idioms and the resulting register name repetition. In Figure 3, the three instruction sequence (**lda**, **ldq**, **cmplt**) increments an array pointer, loads the value, and compares it to a second value. The 7 register names used within this sequence represent four distinct values: the array element pointer, the array element value, the compared value and the comparison result. Given four register parameters, we could generalize this sequence completely. Second, a small number of bits (we use five) suffice to effectively capture most immediate values. Certainly, most immediate fields are wider than five bits. The key here is that in the static uses of a given decompression entry only a few immediates will be used and this small set can be compactly represented in a small number of bits. In our example, the immediate used in **lda** is the size of the array element. In a 64-bit machine, this number will most likely be some small integer multiple of 8. By defining the interpretation of the bits of **P2** to be the three-bit left-shift of the literal parameter, we can capture a large number of array accesses in a single dictionary entry.

3.2 Compression Algorithm

Code compression for DISE consists of three steps. First, a *compression profile* is gathered from one or more applications. Next, an iterative algorithm uses the compression profile to build a *decompression dictionary* (*i.e.*, the virtual contents of the RT). Finally, the static executable is compressed using the dictionary (in reverse) to replace compressible sequences (*i.e.*, those that match dictionary entries) with appropriate DISE codewords. We elaborate on each step, below.

Gathering a compression profile. A *compression profile* is a set of weighted instruction sequences extracted from one or more applications. If customized per-program dictionaries are supported, the compression profile for a given program is mined from its own text. If the dictionary is fixed (*i.e.*, a single dictionary is used for multiple programs), a profile that represents multiple applications may be more useful.

```
1  Initialize dictionary D
2  P ← GenerateCompressionProfile({programs})
3  while ∃p ∈ P s.t. benefit(p) > cost(p)
4        select p ∈ P with largest benefit(p)−cost(p)
5        P ← P − {p}
6        UpdateDictionary(D, p)   { unify p with existing
7                                   entries of D if possible }
8        foreach q ∈ P
9              benefit(q) ← RecalculateBenefit(D, q)
10 return D
```

Figure 4: Dictionary construction algorithm.

A compression profile may contain a redundant and exhaustive representation of instruction subsequences in a program. For instance, the sequence <1,2,3,4> may be represented by up to six sequences in the profile: <1,2>, <2,3>, <3,4>, <1,2,3>, <2,3,4>, and <1,2,3,4>. This exhaustive representation is not required, but it gives the dictionary construction algorithm (below) maximum flexibility, improving resultant dictionary quality. We limit the maximum length of these subsequences to some small k (the minimum length of a useful sequence is two instructions), and we do not allow the sequences to span basic blocks. The latter constraint is necessary for correctness because DISE does not permit control to be transfered to the middle of a replacement sequence. Both constraints limit the size of compression profiles and instruction sequence lengths, which are naturally not very long (see Section 4).

A weight is associated with each instruction sequence in a profile in order to estimate the potential benefit of compressing it. We compute benefit of sequence p via the formula: benefit(p) = weight(p) $\times$ (length(p) $-$ 1). The latter factor represents the number of instructions eliminated if an instance of p is compressed to a single codeword. Weight may be based on a static measure (*i.e.*, the number of times the sequence appears in the static executable(s)), a dynamic measure (*i.e.*, the number of times the sequence appears in some dynamic trace or traces), or some combination of the two, allowing the algorithm to target compression for static code size, reduced fetch consumption—a feature that can be used to reduce instruction cache energy consumption—or both. For best results, the weights in a profile should match the overlap relationships among the instruction sequences. In particular, the weight of a sequence should never exceed the weight associated with one of its proper subsequences, since the appearance of the subsequence must be at least as frequent as the appearance of the supersequence.

Building the dictionary. A compression/decompression dictionary is built from the instruction sequences in a compression profile using the iterative procedure summarized in Figure 4. At each iterative step, the instruction sequence with the greatest estimated compression benefit (minus its cost in terms of space consumed in the dictionary) is identified and added to the dictionary. In environments where the dictionary is fixed and need not be encoded into the application binary, we set the cost of all sequences to zero. In this case, it may be useful to cap the size of the dictionary to prevent it from growing too large. The iterative process continues until no instruction sequences have a benefit that exceeds their cost.

When a sequence is added to the dictionary, corrections must be made to the benefits of all remaining sequences that fully or par-

tially overlap it to account for the fact that these sequences may no longer be compressed. Since DISE only expands the fetch stream and does not re-expand the expanded stream, a sequence that contains a decompression codeword cannot itself be compressed. We recompute the benefit of each sequence (using RecalculateBenefit()) given the sequences that are currently in the dictionary and information encoded in the profile.

Benefit correction monotonically reduces the benefit of a sequence, and may drive it to zero. For example, from our group of six sequences, if sequence <1,2,3> is selected first, the benefit of the sequence <1,2,3,4> goes to zero. Once <1,2,3> is compressed, no sequence <1,2,3,4> will remain. If <1,2,3,4> is selected first, the benefit of sequence <1,2,3> will be reduced, but perhaps not to zero. Once <1,2,3,4> is compressed, instances of <1,2,3> may still be found in other contexts.

Incorporating parameterized compression. The dictionary building algorithm is easily extended to support parameterized compression. At each step, before adding the selected sequence to the end of the dictionary, we attempt to *unify* it via parameterization with an existing entry. Two sequences may be unified if they differ by at most *p* distinct register specifiers or immediate values, where *p* is the maximum number of parameter values that can be accommodated within a given instruction (a 32-bit instruction can realistically accommodate 3). For instance, assuming *p* is 1, the sequence <**addq r2,r2,8; ldq r3,0(r2)**> can be unified with the existing sequence <**addq r4,r4,8; ldq r3,0(r4)**> by the decompression entry <**addq p1,p1,8; ldq r3,0(p1)**>. The sequence <**addq r2,r2,16; ldq r3,0(r2)**> cannot be unified with the existing sequence using only a single parameter. DISE does not support opcode parameterization. If unification is possible, the sequence is effectively added to the dictionary for free, *i.e.*, without occupying any additional dictionary space. If unification with multiple entries is possible—a rare occurrence since it implies that two nearly identical entries were not already unified with each other—the one that necessitates the fewest number of parameters is chosen.

Even in environments where the virtual dictionary size is capped, parameterization allows us to continue to add sequences to the dictionary so long as they can be unified with existing entries. Similarly, the algorithm adds sequences whose cost exceeds their benefit if they may be unified with existing dictionary entries (*i.e.*, they have effectively no cost).

Compressing the program. Given a decompression dictionary—a set of decompression productions and their RT identifiers (virtual indices)—compressing a program is straightforward. The executable is statically analyzed and instruction sequences that match dictionary entries are replaced by the corresponding DISE codewords. The search-and-replace procedure is performed in dictionary order. In other words, for each dictionary entry, we scan the entire binary (or function), and compress all instances of that entry before attempting to compress instances of the next entry. This compression order matches the order implicitly assumed by our dictionary selection algorithm. When compression is finished, branch and jump targets—including those in jump tables and PC-relative offsets in codewords—are recomputed.

4. EXPERIMENTAL EVALUATION

DISE is an effective mechanism for implementing dynamic decompression in both general purpose and embedded processors. We demonstrate this using cycle-level simulation. Our primary metric is compression ratio, the ratio of compressed to uncompressed program size. Section 4.2 shows the effectiveness of DISE-based compression versus a dedicated-hardware approach. Section 4.3 explores the sensitivity of compression to factors such as dictionary size and number parameters. Section 4.4 demonstrates the impact of program-specific compression. The final two sections evaluate (de)compression's performance and energy implications.

4.1 Experimental Environment

Simulator. Our simulation tools were built using the SimpleScalar Alpha instruction set and system call definition modules [5]. The timing simulator models a MIPS R10000-like processor with a parameterizable number of pipeline stages, register renaming, out-of-order execution, aggressive branch and load speculation and a two-level on-chip memory hierarchy. Via parameter choices, we model both general-purpose and embedded cores. The general-purpose core is 4-way superscalar, with a 12 stage pipeline, 128 entry re-order buffer, 80 reservation stations, 32KB instruction and data caches, and a 1MB L2. The embedded configuration is 2-way superscalar, with a 5 stage in-order pipeline, an 8KB instruction cache, 16KB data cache, and no L2. The simulator also models a parameterized DISE mechanism. Our default configuration uses a 32 entry PT (although decompression requires only a single PT entry) and a 2K-instruction 2-way set-associative RT. Each PT entry occupies about 8 bytes while each RT entry occupies about 6 bytes—replacement instruction specifications are represented using 8 bytes in the executable, but this representation is quite sparse—so the total sizes of the two structures are 512 bytes and 12KB, respectively. For the general purpose configuration, we assume a 2-stage decoder, so DISE expansion introduces no overhead. For the embedded configuration, we assume an *a priori* 1-stage decoder. The DISE configurations have an additional pipeline stage and suffer an increased branch misprediction penalty. The DISE interface and its cost are not explicitly modeled. We model the DISE miss handler by flushing the pipeline and stalling for 30 cycles.

The simulator models power consumption using the Wattch framework [4] and CACTI-3 [21]. Our power estimates are for 0.35μm technology. The structures were parameterized carefully to minimize power consumption and roughly mirror the per-structure power distributions of actual processors. For a given logical configuration, CACTI-3 employs both squarification and horizontal/vertical sub-banking to minimize some combination of delay, power consumption and area. We configure both the instruction cache and RT as two-way interleaved, single (read/write) ported structures that are accessed at most once per cycle.

Benchmarks. We perform our experiments on the SPEC2000 integer and MediaBench [15] benchmarks. All programs are compiled for the Alpha EV6 architecture with optimization flags *-O4 -fast*. Our simulation environment extracts all nops from both the dynamic instruction stream and the static program image. Nops are excluded from all measurements. When execution times are reported for SPEC, they come from complete runs sampled at 10% (100M instructions per sample) using the train input. MediaBench results are for complete runs using the inputs provided [15]; no sampling is used.

Dictionaries. Compression profiles are constructed by static binary analysis. The compression tool generates a set of decompression productions (the dictionary) via the algorithm presented in Section 3. Our default compression parameters are a maximum dictionary entry length of 8 instructions and no more than 3 register/immediate parameters per entry. Except for the experiments in Section 4.4, a custom dictionary is used for each benchmark. Except for the experiment in Section 4.6, each dictionary is constructed using a compression profile encoding static instruction sequence frequency.

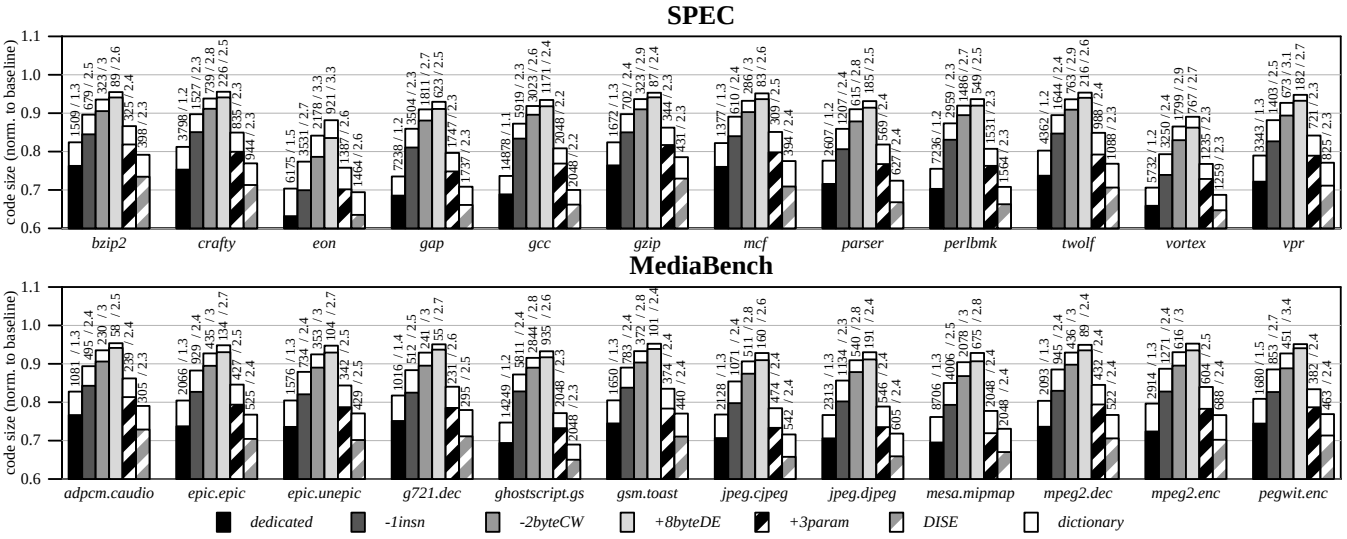

Figure 5: Dedicated and DISE-based feature impact on compression.

4.2 Compression Effectiveness

We begin with a comparison of the compression efficacy of DISE to that of a previously proposed system that exploits dedicated decompression-specific hardware [16]. The dedicated approach does not support parameterized replacement. As a result, it cannot compress PC-relative branches or share dictionary entries in certain situations, but it does have smaller dictionary entries (no directives) and smaller codewords (no parameters), and so it can profitably compress single instruction sequences.

We separate the impact of these differences in Figure 5. Bars represent static compression ratio broken down into two components. The first (the bottom, shaded portion of each stack) is the (normalized) compressed size of the original program text. The second (the top white portion) is the size of the dictionary as a fraction of original program text size. The two numbers written on top of each bar are the total number of dictionary entries, and the number of instructions per entry, respectively. Each bar gives the compression of a decompressor with a slightly different feature set.

Dedicated decompression features. The first bar (*dedicated*) corresponds to a dedicated hardware decompressor, complete with 2-byte codewords and single-instruction compression [16]. The compression ratios achieved—about 70–75% of original text size, dictionary not included (note the scale of the graph)—are comparable to those previously published [16]. In the next two experiments, 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%.

DISE decompression features. With dedicated-decompression-specific features removed, the next three bars add DISE-specific features. The use of parameterized replacement requires four additional bytes per dictionary entry to hold the instantiation directives (*+8byteDE*). Note, this is a highly conservative estimate as there are five fields in a given instruction (opcode, three register specifiers, and an immediate) and each can be modified using five or so different directives. Reserving 32 bits for directives keeps the dictionary section in the executable aligned and provides headroom for future directive expansion. 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 down to 75–80%). 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 the dedicated hardware scheme.

The numbers on top of the bars—number of dictionary entries and average number of instructions per entry—point to interesting differences in dictionary space usage between the dedicated and DISE schemes. While the two schemes use roughly the same amount of total dictionary storage, recall that DISE requires twice the storage per instruction, meaning the DISE dictionaries contain roughly half the number of instructions as the dedicated ones. Beyond that, dedicated dictionaries typically consist of a large number of small entries, including many single-instruction entries. DISE dictionaries typically consist of a smaller number of longer entries. The difference is due to the absence of single-instruction compression—which means that the average compression sequence length must be at least two—and the use of 4-byte codewords which require a longer compressed sequences to be profitable. Parameterized replacement does not increase the average entry size, it just makes more entries profitable since they can be shared among more static locations.

Notice, the total number of dictionary entries for the DISE schemes cannot exceed 2K, since parameterized DISE codewords contain only 11 bits of RT identifier space.

For the remainder of this evaluation, we present results for a representative subset of the benchmarks.

4.3 Sensitivity Analysis

The results of the previous section demonstrate that unconstrained (de)compression is effective. Below, we investigate the impact dictionary entry size (in terms of instructions), total dictionary size, and the number of register/immediate parameters per dictionary entry.

Dictionary entry size. Post-fetch decompression restricts

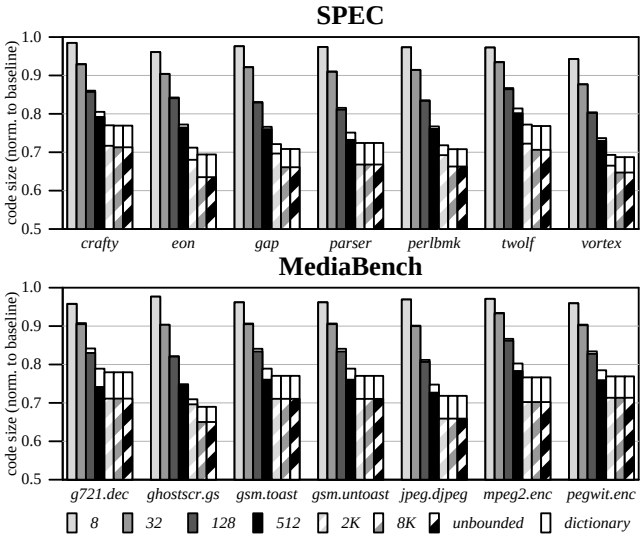

Figure 6: Impact of dictionary size.

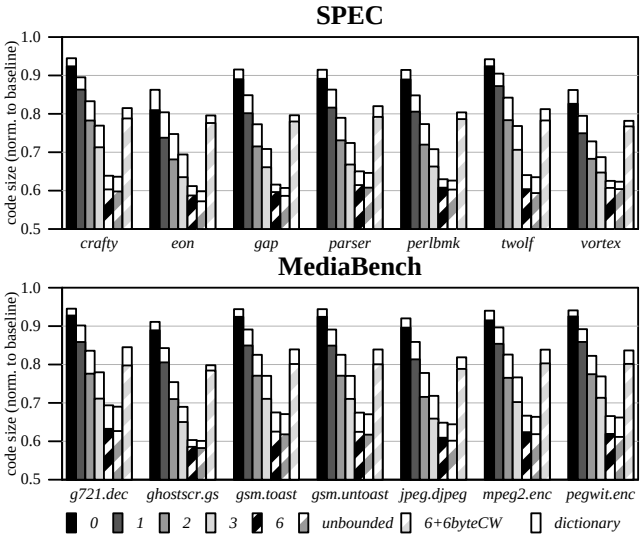

Figure 7: Impact of parameters.

(de)compression sequences to reside fully within basic blocks. Although basic block size is small in the benchmarks we consider, there may be benefit to restricting dictionary entry size even beyond this natural limit. Small blocks may admit more efficient RT organizations and tagging schemes and can reduce the running time of the compressor itself. Our experiments (not graphed here) show that 4-instruction sequences allow better compression (up to 8%) than 2-instruction sequences, 8-instruction sequences occasionally result in slightly better compression still, and 16-instruction sequences offer virtually no advantage over those. Our algorithm simply never selects long instruction sequences for compression because similar long sequences do not appear frequently in the codes we studied.

Dictionary size. Although DISE virtualization allows the dictionary to be larger than the physical RT, a dictionary whose working set exceeds RT capacity will degrade performance via expensive RT miss handling. To avoid RT misses, it is often useful to limit the size of the dictionary, but this naturally degrades compression effectiveness. Figure 6 shows the effect of dictionary size on compression ratio. Note, we define dictionary size as the total number of instructions, *not* the number of entries (*i.e.*, instruction sequences).

Non-trivial compression, reductions of 1–5% in code size are possible with dictionaries as small as 8 total instructions, and 12% reductions are possible with 32-instruction dictionaries (*e.g.*, *vortex*). 512-instruction dictionaries achieve excellent compression, 70–80% of original program code size on all programs. Increasing dictionary size to 2K instructions yields small benefits. Only the larger benchmarks (*i.e.*, *eon*, *perlbmk*, and *ghostscript*) reap additional benefit from an 8K instruction dictionary.

Number of parameters. Parameterized decompression allows for smaller, more effective dictionaries, because similar-but-not-identical sequences can share a single dictionary entry as in Figure 3. This is a feature unique to DISE; Figure 7 shows its impact.

Compression ratios improve steadily as the number of parameters is increased from zero to three; the difference between zero and three parameters is about 15% in absolute terms. Compression improves even further if more than three parameters are used, but there is little benefit to allowing more than six parameters. This diminishing return follows directly from our dictionary entry size results. Each instruction contains no more than three registers (or two registers and one immediate). Since most dictionary entries are 2–4 instructions long, they cannot possibly contain more than 12 distinct register names or immediate values. Of course, in practice the number of distinct names is much smaller. Contiguous instructions tend to be data-dependent and these dependences are represented by shared register names. Parameterized replacement therefore has the nice property that a few parameters capture a significant portion of the benefit. The final bar (*6+6byteCW*) repeats the 6-parameter experiment, but uses longer—6 rather than 4 byte—codewords to realistically represent the overhead of encoding additional parameters. The use of longer codewords makes the compression of shorter sequences less profitable, completely overwhelming the benefit achieved by the additional three parameters. Three parameters—the maximum number that can fit within a 32-bit codeword and still maintain a reasonably sized RT identifier—yields the best compression ratios.

Other experiments (not presented) show that parameterization is slightly more important at small dictionary sizes. This is an intuitive result, as smaller dictionaries place a higher premium on efficient dictionary space utilization.

4.4 Dictionary Programmability

One advantage of DISE (de)compression is dictionary programmability, the ability to use a per-application dictionary. Although previous proposals for post-fetch decompression [16] did not explicitly preclude programmability, a programming mechanism was never proposed and the impact of programmability was never evaluated. In DISE, dictionary manipulation is performed via the controller.

We consider the impact of programmability by comparing three compression usage scenarios. In (*application*) we create a custom dictionary for each application and encode it into the executable. All the data above assumes this scenario. The other two scenarios assume a fixed, system-supplied dictionary, that either resides in kernel memory or is perhaps hardwired into the RT. In these scenarios, the system provides the compression utility. The first of these (*suite*) models a system with a limited but well-understood application domain. Here, we build a dictionary using static profile data collected from the other applications in the benchmark suite.

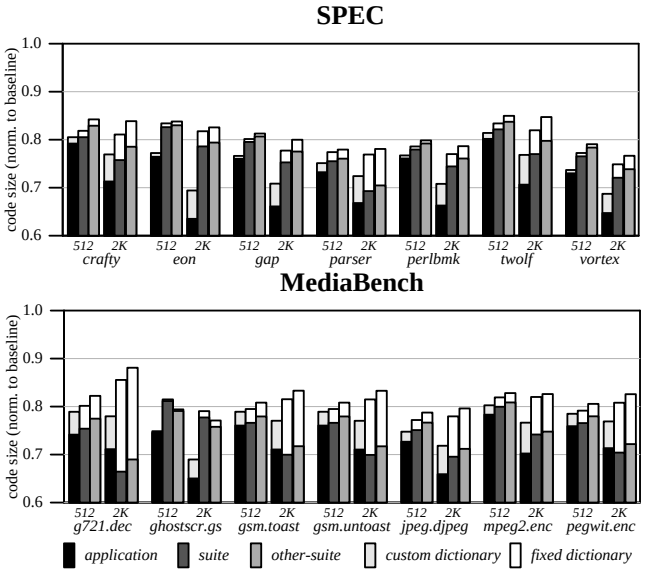

Figure 8: Impact of dictionary customization.

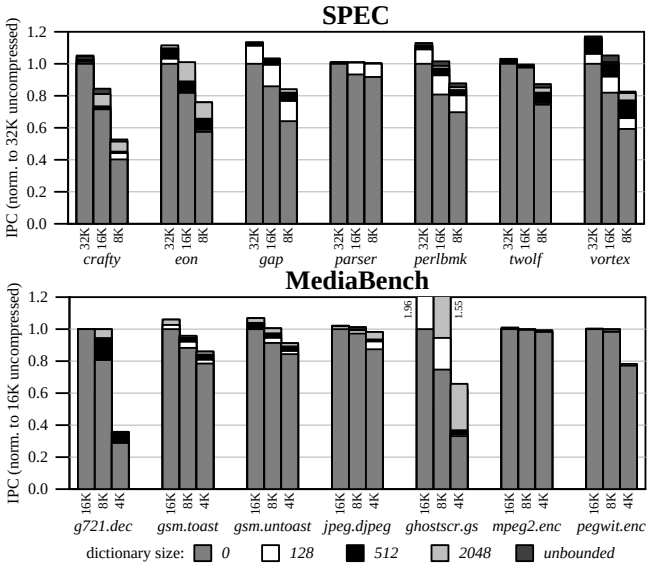

Figure 9: Impact of instruction cache and dictionary.

The second (*other-suite*) models a system with little or no *a priori* knowledge of the application domain. Here, dictionaries are built using profile data from programs in the other benchmark suite. One advantage of system-provided (*i.e.*, fixed) dictionaries is that they do not consume space in the compressed application's executable.

Figure 8 shows the impact of each usage scenario on compression ratio. We actually show the results of two experiments, limiting dictionary size to 512 and 2K total instructions, respectively. There are two interesting results. Not surprisingly, at small dictionary sizes, an application-specific dictionary (*application* outcompresses a fixed dictionary (*suite*, *other-suite*), even when considering that dictionary space is part of the compressed executable in this scenario and not the other two scenarios. Being restricted to relatively few compression sequences while limiting the overall cost of the dictionary to the application places a premium on careful selection and gives the *application* scenario an advantage. As dictionary size is increased, however, careful selection of sequences becomes less important while the fact that entries in fixed dictionaries are "free" to the application increases in importance. With a 2K instruction dictionary, "inversions" in which an application-agnostic dictionary outperforms the application-specific one are observed (*e.g.*, *g721*, *gsm*, *pegwit*). Of course, these are achieved using very large fixed dictionaries which would not be used if the application were forced to include the dictionary in its own binary.

Another note is that the *suite* scenario often outcompresses *other-suite*, implying that there is idiomatic similarity within a particular application domain. For instance, a few of the MediaBench programs have many floating-point operations whose compression idioms will not be generated by the integer SPEC benchmark suite. The one exception to this rule is *ghostscript*, which arguably looks more like an integer program—it's call-intensive in addition to loop-intensive—than an embedded media program.

4.5 Performance Impact

The performance of a system that uses DISE decompression effectively depends on the average access times of two caches: the instruction cache and the RT, which acts as a cache for the dictionary. Since each is accessed in an in-order front-end stage, penalties are taken in series and translate directly into end latency.

The remainder of the evaluation focuses on performance and energy, variations in which are due to tradeoffs between the instruction cache and RT.

Instruction cache performance. Figure 9 isolates instruction cache performance by simulating an ideal DISE engine, an infinite RT with no penalty per expansion. The figure shows the relative performance of fifteen instruction-cache/dictionary configurations: each of three cache sizes used in conjunction with each of five (de)compression dictionary sizes—0 (no decompression), 128 entries, 512, 2K, and an unbounded dictionary. We show performance (IPC; higher bars are better) normalized to a 32KB instruction cache with no (de)compression. Of the three components of average access time—hit time, miss rate, and miss penalty—only the miss rate concerns us; we fix the miss penalty and ignore the fact that smaller caches could be accessed in fewer pipeline stages.

While larger dictionaries can improve static compression ratios, small ones suffice from a performance standpoint. For many programs, much of the static text compressed by larger dictionaries is not part of the dynamic working set, and its compression does not influence effective cache capacity. About half of the programs (*e.g.*, *gap*, *parser*, and *perlbmk*) benefit little from dictionaries larger than 128 total instructions, and only *crafty* and *eon* show significant improvement when dictionary size is increased beyond 2048 instructions.

Counter-intuitively, compression may hurt cache performance by producing pathological cache conflicts that did not exist in uncompressed (or less aggressively compressed) code. This effect is especially likely to occur at small cache sizes. A prime example is *ghostscript*, although not immediately evident from the figure, on the 8KB and 4KB caches, the 512 instruction dictionary actually underperforms the 128 instruction dictionary. The pathological conflict—actually there are two clustered groups of conflicts each involving 4–5 sets—disappears when the larger, 2K instruction dictionary is used. We have verified that this artifact disappears at higher associativities (*e.g.*, 8-way). The same effect occurs, but to a far lesser degree, in *gap* and *twolf*. The presence of such artifacts argues for programmable compression.

DISE engine performance. In contrast with the instruction cache, we are concerned with all aspects of RT performance. RT

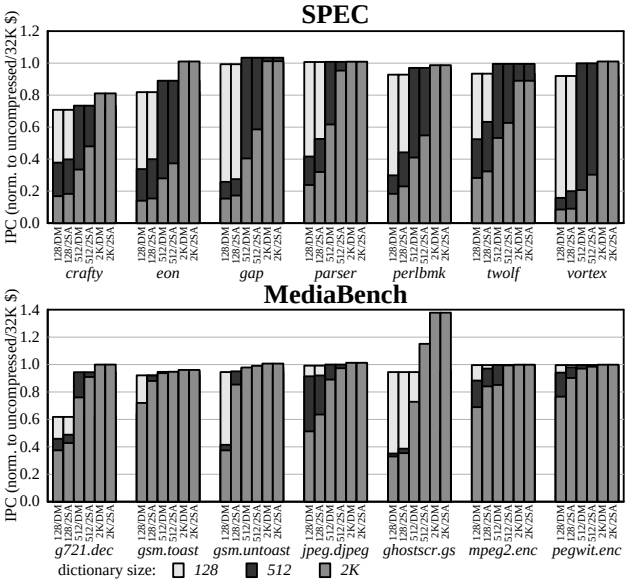

Figure 10: Performance impact of RT misses.

hit time is determined by the DISE engine pipeline organization. The PT and RT are logically accessed in series. In a two-stage decoder, serial PT/RT access could be hidden with no observed penalty. However, adding DISE to a single-cycle decoder requires either an additional pipeline stage which results in a one-cycle penalty on every mispredicted branch or, if the PT and RT are placed in parallel in a single stage, a one cycle penalty on every PT match. Although not shown, the performance of these configurations is quite intuitive. The cost of elongating the pipeline is proportional to the frequency of mispredicted branches in the instruction stream, about 0.5–1%. The cost of a one cycle delay per expansion is proportional to expansion frequency, quite high for ACFs like (de)compression which make heavy use of DISE. While the pipelined approach seems less aesthetically pleasing because it penalizes ACF free code, any system for which heavy DISE use is anticipated—primarily one for which expansion will be more frequent than branch misprediction—should use it.

The other components of RT access time are miss rate and the cost of servicing a miss. The RT miss rate is a function of virtual dictionary working set size and the physical RT configuration, primarily the capacity. RT misses are quite expensive. We model the RT miss penalty by flushing the pipeline and stalling for 30 cycles. Figure 10 shows the performance of systems with several virtual dictionary sizes (128, 512, 2048 instructions) on RTs of several different configurations (128, 512, and 2048 instruction specification slots arranged in four instruction blocks, both direct mapped and 2-way set-associative). Performance is normalized to the "large cache" DISE-free configuration, while the DISE experiments all use smaller caches. For this reason, slowdowns—normalized performance of less than 1—are sometimes observed, especially for the small physical RT configurations. Since the RT miss penalty is fixed, performance differences are a strict function of the RT miss rate.

As the figure shows, a large virtual dictionary on a small physical RT produces an abundance of expensive RT misses which cause frequent execution serializations. A 2K-instruction dictionary executing on a 128 entry RT can degrade performance by a factor of 5 to 10 (*e.g., vortex*). Although RT virtualization guarantees correct execution, to preserve performance, dictionaries should not be allowed to exceed the physical size of the RT. The instruction conflict pathology described in the previous section is again evident in *twolf*. On a 2K-instruction RT, the 512-instruction dictionary outperforms the 2K-instruction dictionary, even though neither generates RT misses.

The MediaBench programs typically require smaller dictionaries and are more loop oriented than their SPEC counterparts. 2K-instruction dictionaries are rare even when no limit is placed on dictionary size, and dictionaries tend to exhibit better RT locality. As a result, larger dictionaries perform relatively better on small RTs than in SPEC.

4.6 Energy Implications

In a typical general purpose processor, instruction cache access accounts for as much as 20% of total processor energy consumption. Other structures, like the data cache and L2 cache, may be as large or larger than the instruction cache, but are accessed less frequently (the instruction cache is accessed nearly every cycle), and typically one bank at a time (all instruction cache banks are accessed on each cache access cycle). In an embedded processor, which may contain neither an L2 nor a complex execution engine, this ratio may be even higher.

Post-fetch decompression can be used to reduce energy consumption, both in the instruction cache and in total. Energy reduction can come from two sources: (i) reduced execution times due to compressed instruction footprints and fewer instruction cache misses, and/or (ii) the use of smaller, lower-power caches. However, there are two complementary sources of energy consumption increase. First, the DISE structures themselves consume energy. Second, the use of a smaller instruction cache may decrease effective instruction capacity beyond compression's ability to compensate for it, increasing instruction cache misses and execution time. These effects must be balanced against one another. The potential exists for doing so on a per-application basis by selectively powering down cache ways [2] or sets [23]. A similar strategy can be used for the RT.

Energy and EDP. Figure 11 shows the relative energy consumptions and energy-delay products (EDP) of several DISE-free and DISE (de)compression configurations. Energy bars are normalized to total energy consumption of the DISE free system with the larger (32KB or 16KB) instruction cache, respectively. Each bar shows three energy components: instruction cache (white), DISE structures (black), and all other resources (gray). Notice, instruction cache energy is about 15-25% of total energy in a general purpose processor and 35-45% in an embedded processor. The EDP for each configuration is shown as a triangle. There are eight total configurations, uncompressed and compressed with three RT sizes for each of two instruction cache sizes. Since RT misses have a high execution time and thus energy cost, we use virtual dictionaries that are of the same size as the physical RTs.

DISE (de)compression can reduce total energy and EDP even though the tradeoff between cache and RT instruction capacity highly favors the cache. In the first place, accessing two 16KB structures consumes more energy than accessing a single 32KB structure. Although wordline and bitline power grows roughly linearly with the number of RAM cells in an array, the power consumed by supporting structures—wordline decoders, sense-amplifiers and output drivers—is largely independent of array size. Multiple structures also consume more tag power. Our simulations show that a 32KB single-cache consumes only slightly over 40% more energy per access than a single-ported 16KB cache, not 100% more. Beyond that, however, an RT is less space efficient than an

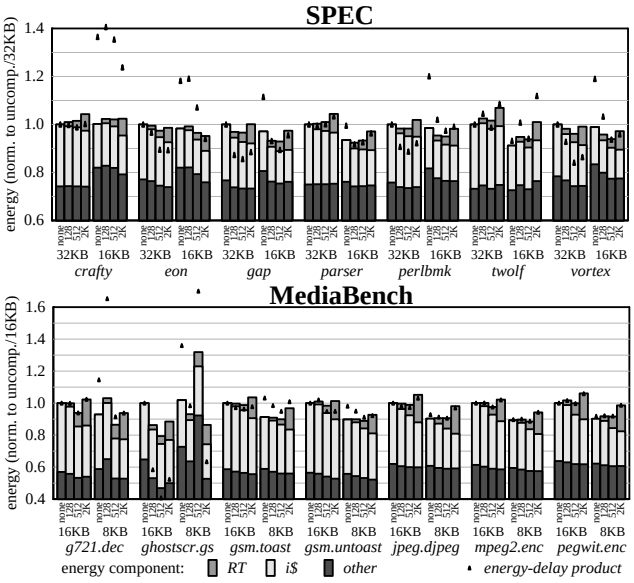

Figure 11: Impact of compression on energy.

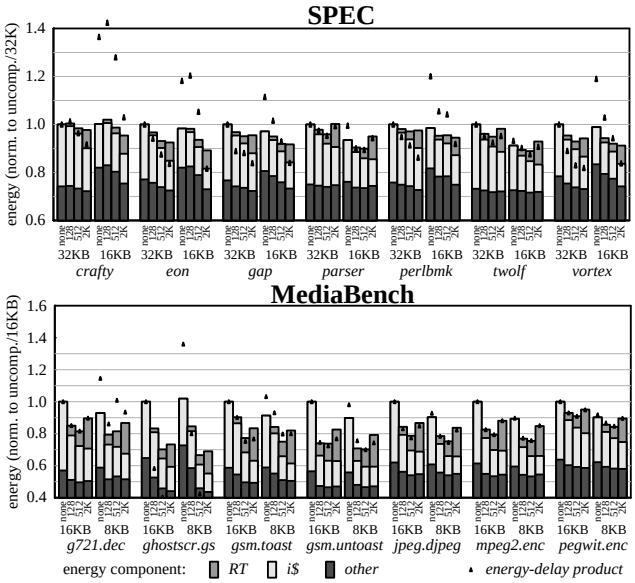

Figure 12: Impact of dynamic code compression on energy.

instruction cache because it must store per-instruction instantiation directives as well. When we combine these factors, we see that in order to save energy over a 32KB configuration, we must replace 16KB of cache (storage for 4K instructions) with a 3KB RT (storage for 512 replacement instruction specifications). Fortunately, the use of parameterized replacement enables even small dictionaries to cover large static instruction spaces, making this organization profitable.

For most benchmarks, the lowest energy (or EDP) configuration combines an instruction cache with an appropriately sized dictionary and RT. Note, the lowest energy and the lowest EDP are often achieved using different configurations. In general, DISE is more effective at reducing EDP than energy, as it trades instruction cache energy for RT energy. Typical energy reductions are 2-5%, although reductions of 18% are sometimes observed (*e.g.*, *ghostscript* with 16KB instruction cache and 512-instruction dictionary). Without RT misses (recall virtual dictionaries are sized to eliminate misses), performance improvements due to instruction cache miss reductions account for EDP reductions which often exceed 10% (*e.g.*, *eon*, *gap*, *perlbmk*, *vortex*) and sometimes reach 60% (*e.g.*, *ghostscript*).

Targeting compression to reduce cache accesses. A third way to reduce instruction cache energy—and thus total energy and EDP—is to reduce the number of instruction cache *accesses*. Conventional *static* compression attempts to reduce instruction cache misses by compressing sequences that appear frequently in the static text. *Dynamic* compression attempts to reduce cache accesses by compressing sequences that appear frequently in the dynamic execution stream. Our compression algorithm easily builds dynamic compression dictionaries. It simply weighs each instruction sequence in a compression profile by an incidence frequency taken from some dynamic execution profile. In Figure 12, we repeat our experiment using dynamic dictionaries. By greatly reducing instruction cache power, especially for larger dictionaries, dynamic (de)compression provides more significant reductions in both energy and EDP. 10% energy reductions are common (*e.g.*, *eon*, *gap*, *vortex*, *ghostscript*, *gsm*) as are 20% EDP reductions. Note, the additional EDP reduction comes from the corresponding energy reduction, not from a further reduction in execution time.

5. RELATED WORK

The large body of work on code compression speaks to the importance of the technique.

Software-based approaches. Traditional static optimizations intended to accelerate execution (*e.g.*, dead-code elimination, common sub-expression elimination, register allocation, *etc.*) often have the side effect of reducing code size [8]. We use an optimized uncompressed baseline in our experiments. *Code factoring* replaces common instruction sequences with calls to procedures containing these sequences. Factoring reduces code size at the expense of increased execution time due to function call overhead [6, 8]. ISA extensions have been proposed to reduce this overhead [19].

Non-executable compressed formats permit more aggressive compression but require an explicit and expensive decompression step before execution. Systems have been proposed for decompressing code at the procedure [13] and cache-line granularities [17]. Although effective in reducing code size, the performance of these systems degrades significantly.

ISA extensions. Certain ISAs (*e.g.*, ARM's Thumb [1] and MIPS16 [14]) support compact code via short-form versions of commonly used instructions. Although there is no significant overhead in decompression itself, performance suffers because the short formats provide a limited register and opcode menu, increasing the number of instructions in short format regions (mode switches are required between short and 32-bit code regions). Furthermore, dense instruction encodings do not exploit repetition of code sequences like coarse-granularity (*i.e.*, multiple instruction) (de)compression schemes. Dense encodings and coarse-granularity (de)compression mechanisms are orthogonal and can be used in conjunction.

Hardware-based approaches. Fill-path decompression is a hardware technique in which compressed code in memory is decompressed by the instruction cache fill unit on a miss. Examples of fill-path decompression include the Compressed Code RISC Processor (CCRP) [22] and IBM's CodePack [12]. Although fill-path decompression schemes necessitate no processor core modifications and incur decompression cost only on instruction cache

misses, they often use sequential and computationally expensive compression techniques (*e.g.*, Huffman), resulting in significant runtime overhead. In addition, they store uncompressed code in the instruction cache, so the cache does not benefit from a compressed image and the hardware must map uncompressed addresses to compressed ones. Finally, the unit of compression is limited to the cache line, so individual instructions (or bytes) must be compressed rather than instruction sequences.

DISE performs post-fetch decompression, allowing the instruction cache to store compressed code while maintaining a single static (compressed) image which does not require address translation structures. Implementations of post-fetch dictionary decompression using custom hardware has been previously proposed. One such system [16] uses a very large dictionary (up to 8K entries, each consisting of one or more instructions) and 16-bit codewords (which admit the compression of single instructions) to achieve impressive code size reductions on PowerPC binaries. Another post-fetch decompression system [18] uses variable length codewords and dictionary-based compression of common instructions (not instruction sequences). Our implementation uses general-purpose hardware, a small dictionary, and supports both parameterized and programmable decompression. Although not a fundamental limitation of DISE, our scheme currently uses only 32-bit word-aligned codewords.

Operand factorization [3] extends post-fetch decompression. Building on the observation that compressing whole instructions—*i.e.*, opcodes and operands together—limits the efficacy of a compression algorithm, operand factorization compresses opcodes (tree patterns) and operands (operand patterns) separately. After fetch, tree and operand patterns are decompressed and reassembled to form machine instructions. Operand factorization is effective for very large dictionaries. Via register/immediate parameterization, DISE supports a limited form of operand factoring within the framework of an existing mechanism.

6. CONCLUSION

Code (de)compression is an important tool for architects of both embedded and general purpose microprocessors. In this paper, we present an implementation of dynamic code decompression based on *dynamic instruction stream editing* (DISE), a programmable decoding facility that allows an application's instruction fetch stream to be transformed in a general way to add functionality to the original program [7]. A DISE implementation of (de)compression has many advantages. It implements post-fetch decompression, allowing the instruction cache to benefit from a compressed program image and removing the need for mechanisms for translating uncompressed addresses to compressed ones. DISE's matching and parameterized replacement functionality supports parameterized (de)compression, enabling better dictionary space utilization. DISE's programming interface also allows individual applications to exploit custom (de)compression dictionaries. DISE's most compelling advantages, however, have to do with the fact that DISE itself is an essentially existing mechanism that has nothing to do with decompression as such. The core DISE engine consists of well-studied and highly efficient structures that already exists in current processors. The DISE mechanism has many applications beyond code decompression, making its inclusion in a system design easier to justify, and allowing decompression to be composed with other added functionality.

Our experiments using cycle-level simulation of the MediaBench and SPEC2000 integer benchmarks show that DISE enables code size reductions of 25% to 35% and performance improvements of 5-20%. In addition, we evaluate the decompression features unique to DISE, and we find that parametrization and programmability provide significant compression advantages. Parametrization improves code compression by up to 20% and solves the problems associated with compressing PC-relative branches. Application-customized dictionaries enable better compression than a fixed dictionary built to support a large number of applications. Finally, we show that DISE-based compression can reduce total energy consumption by 10% and the energy-delay product by as much as 20%.

7. ACKNOWLEDGMENTS

The authors thank Vlad Petric for his help with the energy simulations and the anonymous reviewers for their comments. Amir Roth is supported by NSF CAREER award CCR-0238203.

8. REFERENCES

[1] Advanced RISC Machines Ltd. *An Introduction to Thumb*, Mar. 1995.

[2] D. Albonesi. Selective cache ways: On demand cache resource allocation. In *Proc. 32nd International Symposium on Microarchitecture*, pages 248–259, Nov. 1999.

[3] G. Araujo, P. Centoducatte, and M. Cortes. Code compression based on operand factorization. In *Proc. 31st International Symposium on Microarchitecture*, pages 194–201, Dec. 1998.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. 27th International Symposium on Computer Architecture*, pages 83–94, Jun. 2000.

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

[6] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[7] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proc. 30th International Symposium on Computer Architecture*, Jun. 2003.

[8] S. K. Debray, W. Evans, R. Muth, and J. D. Sutter. Compiler techniques for code compression. *ACM Transactions on Programming Languages and Operating Systems*, 22(2):378–415, Mar. 2000.

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

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

[11] L. Gwenapp. P6 microcode can be patched. *Microprocessor Report*, 11(12), Sept. 1997.

[12] T. M. Kemp, R. K. Montoye, D. J. Auerback, J. D. Harper, and J. D. Palmer. A decompression core for PowerPC. *IBM Systems Journal*, 42(6):807–812, Nov. 1998.

[13] D. Kirovski, J. Kin, and W. Mangione-Smith. Procedure based program compression. In *Proc. 30th International Symposium on Microarchitecture*, pages 204–213, Dec. 1997.

[14] K. Kissell. *MIPS16: High-Density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[15] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing

multimedia and communications systems. In *Proc. 30th International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[16] 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.

[17] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Proc. 6th International Symposium on High-Performance Computer Architecture*, pages 218–227, Jan. 2000.

[18] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *Proc. 36th Design Automation Conference*, pages 294–299, Jun. 2000.

[19] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electrical Systems*, 4(1):12–38, Jan. 1999.

[20] T. Szymanski. Assembling code for machines with span dependent instructions. *Communications of the ACM*, 21(4):300–308, Apr. 1978.

[21] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, DEC Western Research Laboratory, 1994.

[22] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. 25th International Symposium on Microarchitecture*, pages 81–91, Dec. 1992.

[23] S.-H. Yang, M. Powell, B. Falsafi, and T. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proc. 8th International Symposium on High Performance Computer Architecture*, Jan. 2002.