The Implementation and Evaluation of Dynamic Code Decompression Using DISE

MARC L. CORLISS, E. CHRISTOPHER LEWIS, and AMIR ROTH
University of Pennsylvania

Code compression coupled with dynamic decompression is an important technique for both embedded and general-purpose microprocessors. *Postfetch 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 postfetch decompression using a new hardware facility called *dynamic instruction stream editing* (DISE). DISE provides a programmable decoder—similar in structure to those in many IA-32 processors—that is used to add functionality to an application by injecting custom code snippets into its fetched instruction stream. We present a DISE-based implementation of postfetch decompression and show that it naturally supports customized program-specific decompression dictionaries, enables parameterized decompression allowing similar-but-not-identical instruction sequences to share dictionary entries, and uses no decompression-specific hardware. We present extensive experimental results showing the virtue of this approach and evaluating the factors that impact its efficacy. We also present implementation-neutral results that give insight into the characteristics of any postfetch decompression technique. Our experiments not only demonstrate significant reduction in code size (up to 35%) but also significant improvements in performance (up to 20%) and energy (up to 10%).

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

General Terms: Performance, Design, Experimentation

Additional Key Words and Phrases: Code compression, code decompression, dynamic instrumentation, dynamic instruction stream editing, DISE

1. INTRODUCTION

Code compression coupled with dynamic decompression is a useful technique in many computing contexts. Certainly, (de)compression benefits embedded

A preliminary version of this work was presented at the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03), June 2003 [Corliss et al. 2003b].
The work was funded in part by NSF grant CCR-03-11199. Amir Roth is supported by NSF CAREER Award CCR-0238203, and E. Lewis is supported by NSF CAREER Award CCF-0347290.
Authors' address: Department of Computer and Information Science, University of Pennsylvania, 3330 Walnut St., Philadelphia, PA 19104-6389; email: {mcorliss,lewis,amir}@cis.upenn.edu.

devices where power, size, and cost constraints force the use of small caches and memories. But general-purpose systems can also 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 [Kemp et al. 1998; Wolfe and Chanin 1992]. The advantages of fill-path decompression is that it allows the use of unmodified cores while incurring the decompression penalty only on instruction cache misses. 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 instruction cache and pipeline) 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 [Lefurgy et al. 1997]. *Postfetch 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 and evaluate an implementation of postfetch code decompression via *dynamic instruction stream editing* (DISE) [Corliss et al. 2003a]. DISE is a hardware-based instruction macroexpansion facility similar in structure and function to IA-32 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, and so on) 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 IA-32 microprocessors [Diefendorf 1998; Glaskowsky 2000; Gwenapp 1997].

A DISE implementation of dictionary-based postfetch decompression has several important virtues. DISE's macroexpansion 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. Parameterization also allows 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 can implement many other features and even combine them (dynamically) with decompression.

We evaluate DISE decompression using custom compression tools and 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 a DISE implementation of hardware decompression—accounts for a significant portion of total compression. We also show that dictionary programmability is an important consideration for dynamic decompression, as well as how to reduce the overhead of application-customized dictionaries. Although previous postfetch decompression proposals do not preclude programmability and may even assume it, none evaluates its importance or provides a mechanism for its implementation. We show that DISE-based compression can reduce total energy consumption by 10% and the energy-delay product by as much as 20%. Finally, we find that the choice of compiler, optimizations, and ISA used to produce a program can impact its compressibility.

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. Section 4 gives an extensive evaluation of DISE-based compression, but many of the results apply to other approaches to code compression. The final two sections summarize related work and conclude.

2. DISE

Dynamic instruction stream editing (DISE) [Corliss et al. 2002, 2003a] is a hardware 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. Software solutions inject instructions into the application's instruction stream, but require expensive binary modification. Hardware approaches typically customize the application using dedicated pipeline stages, but are functionally rigid. DISE is a cooperative software–hardware mechanism for implementing ACFs. 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 macroexpands 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 IA-32 microprocessors [Diefendorf 1998; Glaskowsky 2000; Gwenapp 1997]. 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 macroexpansion is used to simplify the execution engines of CISC processors. DISE serves a different purpose (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 most salient to decompression.

As shown in Figure 1, the DISE hardware complex comprises two blocks (shaded). The *DISE engine* performs matching and expansion of an application's

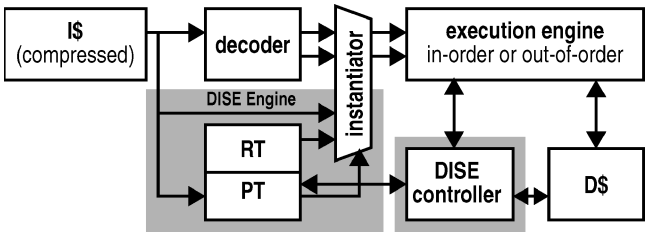

Fig. 1. DISE structures in processor pipeline.

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 IA-32 microprocessors to convert CISC ISA instructions to internal RISC microinstruction sequences [Diefendorf 1998; Glaskowsky 2000; Gwenapp 1997]. The *DISE controller* provides an interface (via the execution engine) for programming the PT and RT, abstracting the microarchitectural formats of patterns, and replacement instructions from DISE clients. While the DISE engine is continuously active (unless disabled), the DISE controller is a coprocessor 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 pattern 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

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

Fig. 2. Sample DISE PT/RT entries.

PT

OP	R1	R2	RD	IMM	RTID
res1	–	–	–	–	–

RT

ID	LITERAL	DOP	DR1	DR2	DRD	DIMM
0	ldq –, 0(–)	–	R1	–	RD	–
0	addi –, 1, –	–	RD	–	RD	–
8	ldq –, 0(–)	–	R1	–	RD	–
8	subi –, 1, –	–	RD	–	RD	–

Fig. 3. Application-aware sample DISE PT/RT entries.

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 (as in Figure 2). 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 (illustrated in Figure 3), 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 original 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. As a result, the example in Figure 3 can be used to produce an instruction sequence that increments (RT identifier 0) or decrements (8) a value in memory.

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 PT) virtualization is 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 mechanisms. 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 faults)—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 multiple 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 [Corliss et al. 2003a]. 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 consider it further here.

DISE Performance. The performance impact of the DISE facility depends on its implementation. A complete discussion is beyond the scope of the present work and is available elsewhere [Corliss et al. 2002, 2003a]. Here we give a brief overview. Three aspects of DISE directly impact performance: (i) the expansion process itself, (ii) demand loading the PT/RT, and (iii) interacting with the DISE controller. For compression, minimal interaction with the controller is required (only at application startup), so this cost is nearly nil. The costs of demand loading the PT/RT are comparable to TLB misses, and we evaluate this in Section 4.5.

The cost of the expansion process itself is very important, because each fetched instruction may potentially need to be expanded. The implementation of the PT and RT must not lengthen the processor cycle time. Systems with relatively slow cycle times (e.g., some embedded and low-power systems) will be able to access both the PT and the RT in a single cycle. Faster systems will need to distribute PT and RT access across two cycles. This can be done for free in processors with 2-stage decoders, but it will introduce delay on expansion in processors with 1-stage decoders.

3. (DE)COMPRESSION WITH DISE

DISE enables an implementation of dictionary-based postfetch decompression that is functionally similar to a previously described scheme [Lefurgy et al. 1997]. The DISE implementation is unique among hardware decompression schemes 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 [Lefurgy et al. 1997]. 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 perapplication basis. For instance, in a 32-bit ISA with 6-bit opcodes, we could use 2K decompression entries (11 identifier bits) and up to three 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 two 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 4. Part (a) shows uncompressed static code; the two boxed three-instruction sequences

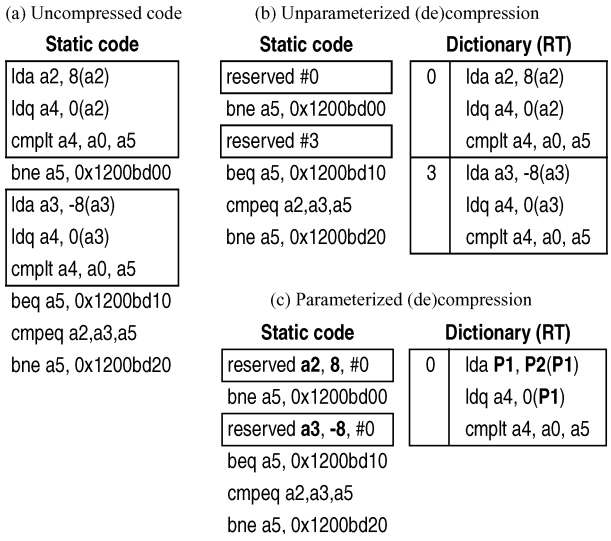

Fig. 4. (De)compression examples.

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 [Szymanski 1978]. In DISE, 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 4, the three-instruction sequence (**lda**, **ldq**, **cmplt**) increments an array pointer, loads the value, and compares it to a second value. The seven 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 5 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 the most common 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. The weight of each sequence represents its static or dynamic frequency. If customized perprogram 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.

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 shared by all existing postfetch decompression mechanisms and 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 the benefit of

```
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
```

Fig. 5. Dictionary construction algorithm.

sequence p via the formula: benefit(p) = weight(p) × (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 (see Section 4.6)—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 outlined in Figure 5. 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. Otherwise, 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 partially 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 reexpand 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.

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 (our implementation actually supports 3), 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. We do not attempt opcode parameterization. If unification is possible, the sequence is effectively added to the dictionary for free, that is, 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.

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. In other words, 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).

Custom, Fixed, and Hybrid Dictionaries. When the compression profile used by this algorithm is derived from a single program, a dictionary will be generated that is customized to its characteristics, resulting in very effective compression for that program. Unfortunately, the dictionary itself must be encoded in the program binary, for it should only be used to decompress that particular program. If a compression profile is derived from a large collection of programs, a dictionary of more general utility is produced, and it may be used to compress a greater variety of programs. Although the effectiveness of this (fixed) dictionary is likely to be inferior to that of a custom dictionary, the dictionary itself need not be encoded in the program binary, because the system (e.g., OS or hardware vendor) can provide this dictionary to be shared by all programs. It may also be valuable to build a hybrid dictionary that includes fixed and customized components, only the latter of which needs to be embedded in the program binary. This hybrid dictionary offers the promise of achieving the best of both customized (good compression of the program itself) and fixed (little overhead) dictionaries. We evaluate all three approaches in Section 4.4.

Rather than compute hybrid custom/fixed dictionaries directly, we combine a custom and fixed dictionary after they have been produced by the algorithm in Figure 5. Assuming a fixed-size virtual RT, we allocate a portion of this

structure to contain fixed dictionary entries (i.e., entries derived from profiling a large class of applications) and the remainder is devoted to custom entries (i.e., entries derived from the particular application being compressed). The former will be shared by all compressed programs, so it need not be encoded in the binary. The latter is unique to each compressed program, so it must be encoded in the program binary. The fixed and custom portions are computed as described above, except that dictionary entries appearing in the custom portion that are *subsumed* by entries appearing in the fixed portion are removed. One entry subsumes another when both are identical except for fields in the subsuming sequence that are parameterized where the subsumed sequence was literal. A (perhaps) superior approach would be to detect and eliminate subsumed entries in the algorithm in Figure 5, but we leave this for future work.

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.

Complexity. Dictionary construction dominates the computational complexity of compression. Because sequences are limited to a maximum constant length (k, above), there are $O(n)$ instruction sequences in the compression profile associated with a program containing (before compression) n instructions. Dictionary construction is quadratic in the number of sequences in the compression profile, so it is quadratic in the size of the uncompressed program. No effort has been applied to optimizing the complexity or performance of the compression algorithm. Nevertheless, for most of our benchmarks compression takes less than 30 s on 2 GHz Pentium 4. We are currently investigating more efficient compression algorithms.

4. EXPERIMENTAL EVALUATION

DISE is an effective mechanism for implementing dynamic decompression in both general-purpose and embedded processors. We demonstrate this using custom tools that implement DISE-based compression. Our primary metric is *compression ratio*, the ratio of compressed to uncompressed program sizes. 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 instruction set architecture, dictionary size, and number of available parameters. Section 4.4 assesses both program-specific and fixed-dictionary compression as well as a hybrid of the two. Sections 4.5 and 4.6 use cycle-level simulation to evaluate the performance and energy implications of

executing compressed code. The final section evaluates the impact of the choice of compiler and optimizations on the compression ratio.

The experimental data presented in this section serves three purposes. First, it demonstrates that DISE-based decompression is effective and evaluates the impact of DISE-specific features (e.g., the impact of parameters on compression ratio and the impact on performance of demand loading the decompression dictionary into the RT). Second, it compares DISE-based decompression with a dedicated-hardware approach. Finally, some of this data (e.g., impact of dictionary size, impact on energy, impact of source compiler, and so on) is, in fact, DISE neutral, so our results are equally relevant to dedicated-hardware implementations.

4.1 Experimental Environment

Simulator. Our results include dynamic program characteristics (e.g., execution time and energy) for which we require simulation of a processor providing the DISE facility. Our simulation tools are built using the SimpleScalar Alpha instruction set and system call definition modules [Burger and Austin 1997]. The timing simulator models an unclustered Alpha 21264 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, 32 KB, 2-way, 1-cycle-access instruction and data caches, and a unified 1 MB, 8-way, 12-cycle-access L2. Memory latency is 100 cycles. It also includes a 6 KB hybrid bimodal/g-share predictor and a 2K entry BTB. The embedded configuration is 2-way superscalar, with a 5-stage in-order pipeline, an 8 KB, 2-way instruction cache, 16 KB 2-way data cache (both 1 cycle access), and no L2. Memory latency is 50 cycles. It includes a 0.75-KB hybrid predictor and a 256-entry BTB.

The simulator also models the 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 12 KB, 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. Here, the DISE-enhanced configuration requires an additional pipeline stage and suffers an increased branch misprediction penalty. The DISE interface and its cost do not impact the use of DISE for code decompression, so they 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 [Brooks et al. 2000], a widely used research tool for architecture power analysis, and CACTI-3 [Wilton and Jouppi 1994], a cache area, access and cycle time, and power consumption estimation tool. Our power estimates are for 0.13 μm

technology. The structures were configured carefully to minimize power consumption and roughly mirror the perstructure power distributions of actual processors. For a given logical configuration, CACTI-3 employs both squarification and horizontal/vertical subbanking to minimize some combination of delay, power consumption, and area. We configure both the instruction cache and RT as 2-way interleaved, single-ported (read/write) structures that are accessed at most once per cycle.

Benchmarks. We perform our experiments on the SPEC2000 integer and MediaBench [Lee et al. 1997] benchmarks. The SPEC benchmarks run on the general-purpose processor configuration, while the MediaBench codes run on the embedded configuration. All programs are compiled for the Alpha EV6 architecture with the native Digital Unix C compiler with optimization flags *-O4 -fast* (except in Section 4.3 in which we evaluate compression for the ARM architecture and Section 4.7 in which we evaluate compression for different compilers and optimization levels). Our simulation environment extracts all nops from both the dynamic instruction stream and the static program image. They are inserted by the Alpha compiler to optimize two idiosyncratic aspects of the Alpha micro-architecture (cache-line alignment of branch targets and clustered execution engine control). Our simulator does not model these idiosyncrasies, so for us the nops serve no purpose and their presence may unfairly exaggerate the benefits of compression. 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 [Lee et al. 1997]; no sampling is used.

Dictionaries. Compression profiles are constructed by static binary analysis (except in Section 4.6). 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 eight instructions and no more than three 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 where weights encode static instruction sequence frequency.

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 [Lefurgy et al. 1997]. 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 6. Bars represent static compression ratio broken down into two components. The first (bottom, shaded portion of each stack) is the (normalized) compressed size of the original

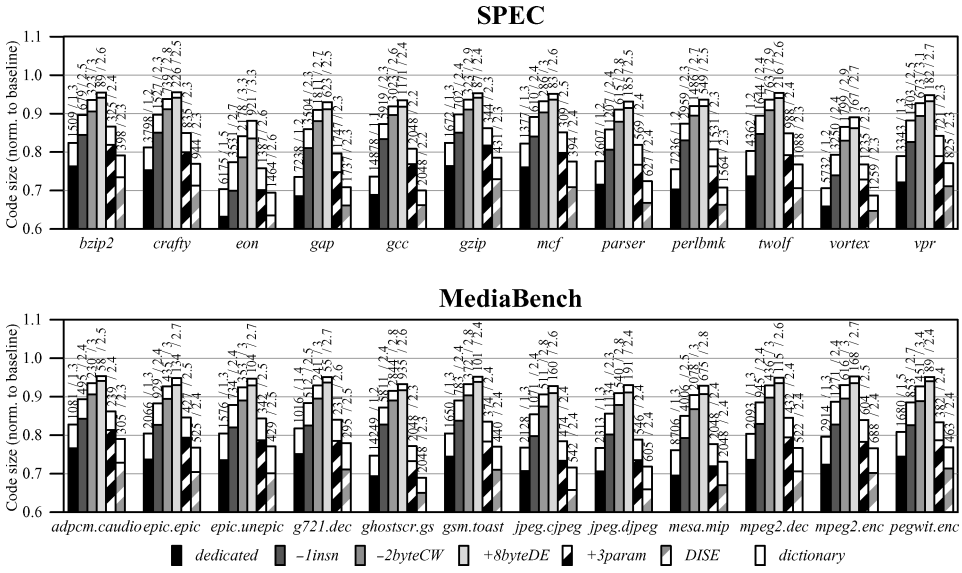

Fig. 6. Dedicated and DISE-based feature impact on compression.

program text. The second (top white portion) is the size of the dictionary as a fraction of original program text size. The combination of these two bars represents the total amount of data required to represent a compressed program. The two numbers written on top of each bar are the total number of dictionary entries, and the average 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 [Lefurgy et al. 1997]. 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 [Lefurgy et al. 1997]. 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 approximately 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 that 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 small 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 (see the portion of each bar), 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 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.

Note that 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 of instruction set architecture, dictionary entry size (in terms of instructions), total dictionary size, and the number of register/immediate parameters per dictionary entry.

Instruction Set Architecture. A program's compressibility naturally depends on the characteristics of the instruction set architecture (ISA) used to represent it. While the Alpha is a high-performance workstation architecture, the ARM architecture is designed to meet the needs of embedded real-time systems, open platforms, and secure applications [Cormie 2002]. The ARM ISA provides a somewhat denser encoding than traditional RISC ISAs (note that we are not evaluating the ARM Thumb compressed representation described in Section 5). Here we investigate the compressibility of programs built from the ARM ISA versus the Alpha ISA. Because ARM was designed for embedded systems, we limit our analysis to the MediaBench codes.

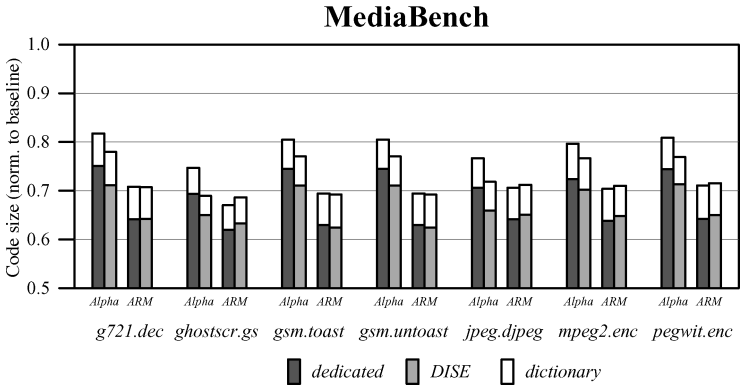

Fig. 7. Alpha versus ARM ISA impact on compression.

Figure 7 presents compression ratios for both Alpha and ARM versions of the benchmarks exploiting both dedicated-hardware and DISE-based approaches to decompression. The compression ratios for both dedicated and DISE-based compression are actually better for the ARM codes. The dedicated-hardware case improves in large part because the ARM versions of these benchmarks have a significantly higher occurrence of a small class of memory operations. The dedicated hardware can compress single instructions, so these memory operations are very effectively compressed. In fact, the number of single instruction sequences grows by a factor of 3 to 5 for each of these codes, and the average sequence length drops dramatically. In addition, our manual analysis of ARM programs reveals that they are more regular than Alpha programs, resulting in better compression for both dedicated and DISE compression. This regularity arises because there are fewer registers (16 versus 32), and a smaller number of opcodes represent a larger portion of the total instruction mix. Because DISE does not benefit from the compression of the greater number of singleton instructions, the dedicated case generally improves more than DISE. Although DISE was more effective for the Alpha, dedicated and DISE-based decompression appear equivalent (in terms of compression ratio) for the ARM ISA. A final factor contributing to compression-ratio differences between Alpha and ARM programs is the compiler used to build the binaries. We used the native Digital Unix compiler to build Alpha binaries and GCC version 2.95.2 (cross compiler) for the ARM binaries. In Section 4.7 we show that different compilers produce code of different compressibility, although this effect is small and certainly does not account for the large differences in Figure 7.

Dictionary Entry Size. Postfetch decompression restricts (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 sequences 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

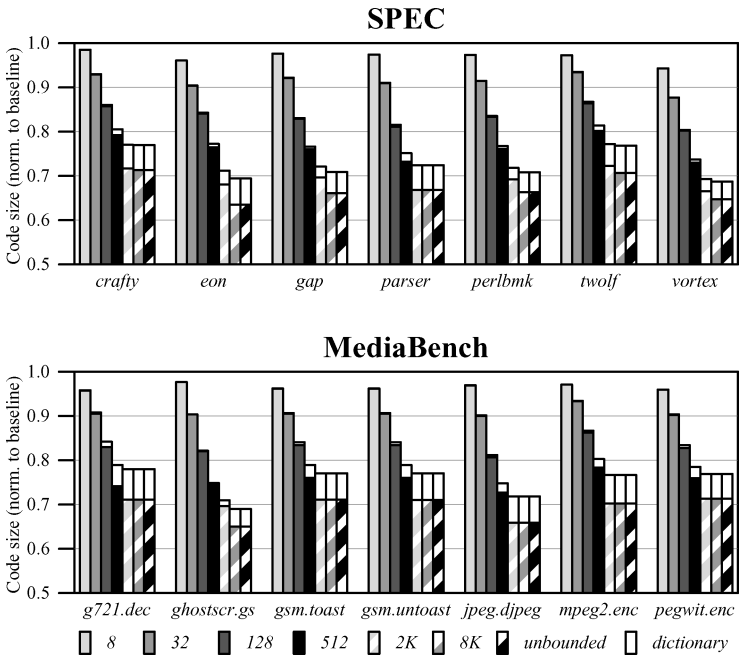

Fig. 8. Impact of dictionary size.

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 8 shows the impact of dictionary size on compression ratio. Note that we define dictionary size as the total number of instructions, *not* the number of entries (i.e., instruction sequences).

Nontrivial compression reductions of 1–5% in code size are possible with dictionaries as small as eight 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. The remaining benchmarks are unable to exploit the additional capacity.

Number of Parameters. Parameterized decompression allows for smaller, more effective dictionaries, because similar-but-not-identical sequences can

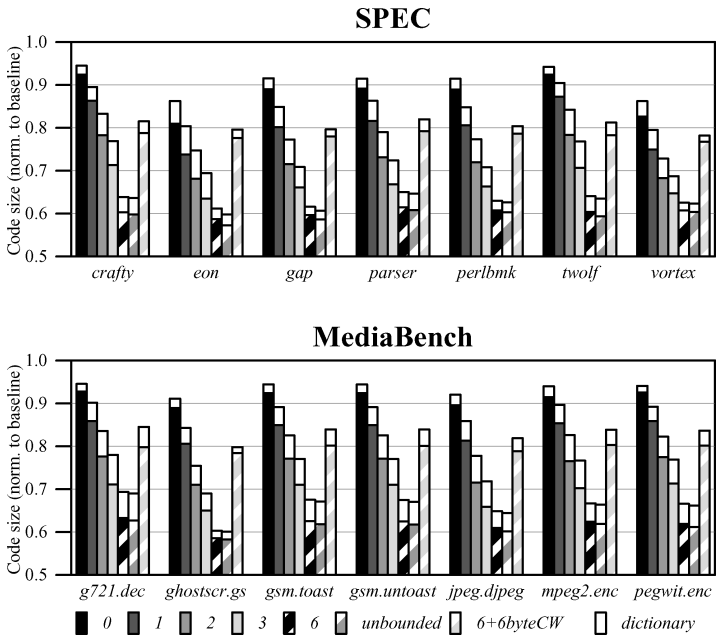

Fig. 9. Impact of parameters.

share a single dictionary entry as in Figure 4. This is a feature unique to DISE among hardware decompression schemes; Figure 9 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 expressed 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—yield 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.

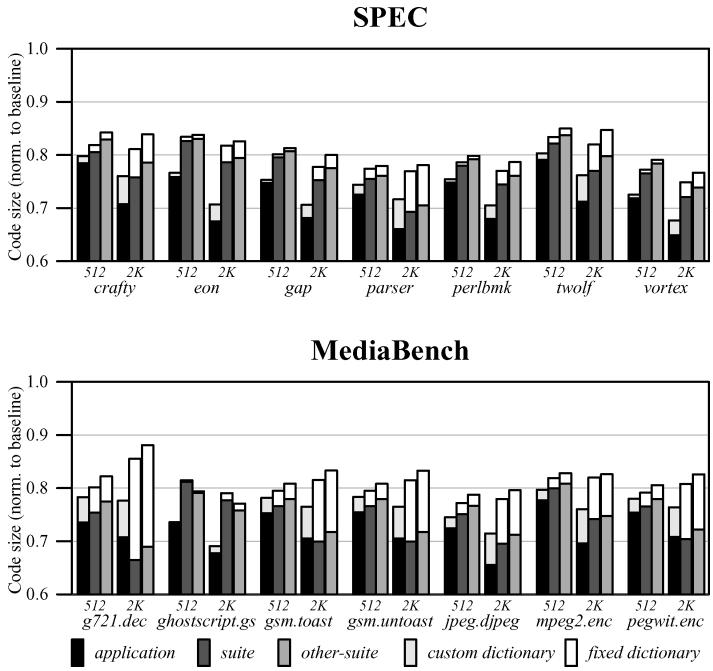

Fig. 10. Impact of dictionary customization.

4.4 Dictionary Programmability

One advantage of DISE (de)compression is dictionary programmability, the ability to use a perapplication dictionary. Although previous proposals for postfetch decompression [Lefurgy et al. 1997] did not explicitly preclude programmability, a programming mechanism was never proposed and the impact of programmability was never evaluated. In DISE, dynamic dictionary manipulation is possible via the controller.

Custom Versus Fixed Dictionaries. 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 presented thus far 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. 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 10 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. Not surprisingly, at small dictionary sizes, an application-specific dictionary (*application*) outcompresses a fixed dictionary (*suite* and *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.

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 is call-intensive in addition to loop intensive—than an embedded media program.

Hybrid Custom/Fixed Dictionaries. From the data in Figure 10, it is apparent that there are unique virtues to both customized (*application*) and fixed (*suite* and *other-suite*) approaches to building and using dictionaries. Customized dictionaries allow for the best compression of the program, but the dictionary itself must be encoded in the program binary, sometimes negating the benefit of customization (e.g., *g721.dec*). Fixed dictionaries have the benefit that they need not be represented in the program binary, but they usually result in poorer compression of the program itself (although this is not always true for reasons described above). A *hybrid* approach for dictionary construction attempts to achieve the best of both worlds.

Figure 11 presents the impact of hybridization. We partition both 512 and 2K entry RTs (the RT must house *both* the custom and fixed part of the dictionary) in six incrementally different ways. The percent under each bar indicates the portion of the total dictionary devoted to custom entries. 100% is completely custom, and 0% is entirely fixed. The white (top) portion of each bar represents the custom portion of the dictionary that must be encoded in the binary. So that all benchmarks share exactly the same fixed portion, our fixed entries are derived from all of the benchmarks within each benchmark suite (including the benchmark being compressed). As a result, the 0% figures differ slightly from the *suite* bars (which exclude the benchmark being compressed) in Figure 10.

Ignoring custom dictionary overhead for the moment (top white portion of each bar), each set of bars exhibits one of three basic shapes: (1) compression ratio (gray bar only) increases as less of the dictionary is dedicated to custom entries, (2) compression ratio decreases, or (3) compression ratio decreases,

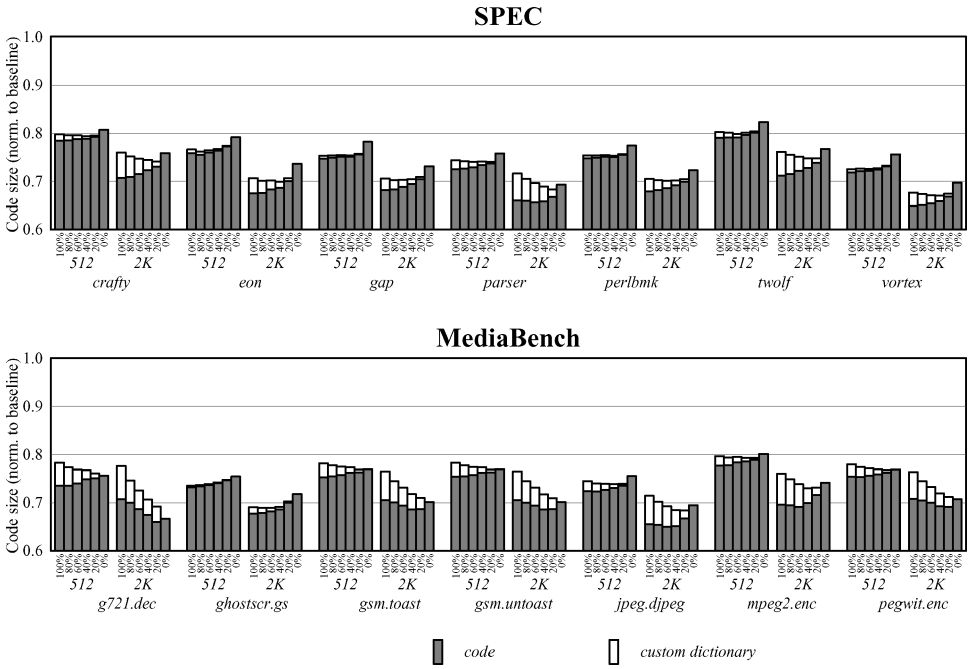

Fig. 11. Impact of hybrid custom/fixed dictionary.

then increases. The first case (e.g., *crafty*, 2K-entry RT) is most natural and common. As the total dictionary becomes less customized to the application being compressed, the compression ratio will naturally get worse (i.e., increase). We see this in almost all SPEC benchmarks and most of the MediaBench codes with small (i.e., 512-entry) RTs.

The second case (compression ratio actually improves when more of the dictionary is devoted to fixed entries) is, at first, unintuitive. The origin of this odd occurrence is that an entry is only added to the custom dictionary if the compression benefit (in the program) exceeds the cost of adding the entry to the dictionary. As a result, it is often the case (particularly for large RTs) that the custom portion of the dictionary is not full, and converting custom entries to fixed entries does not in fact reduce the number of custom entries but it does increase the number of fixed entries. The end result is that there are actually more total entries in the dictionary resulting in better compression. This most naturally occurs for small programs and large dictionaries, so we see it for a number of MediaBench codes with 2K RTs.

The third case (compression ratio improves, then degrades, e.g., *g721.dec*/2K) is the natural combination of the first two cases. The ratio improves at first because fixed-dictionary entries are being added without impacting the custom entries, but at some point, the fixed dictionary cuts into valuable custom entries, degrading compression ratios.

Now we consider the overhead of the custom dictionary (top white portion of each bar). Naturally, this decreases as more of the dictionary is devoted to fixed

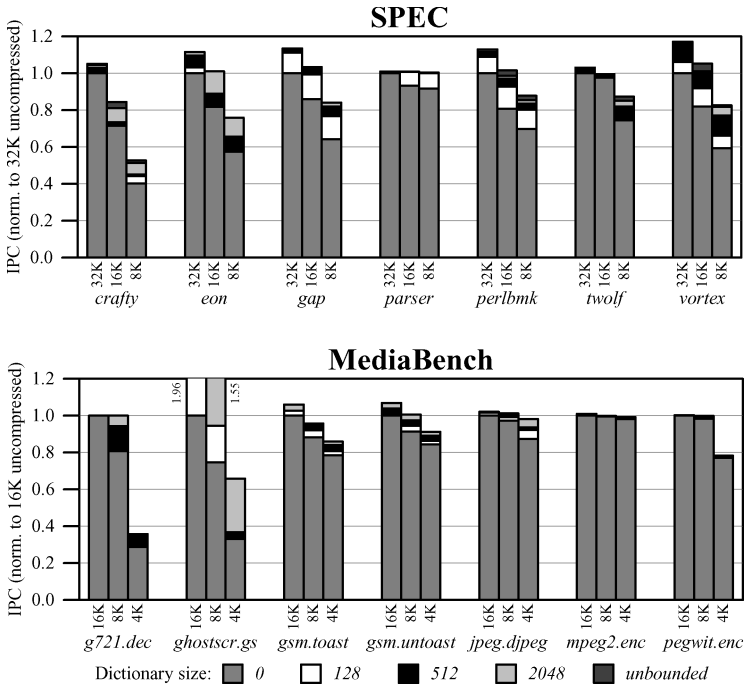

Fig. 12. Impact of instruction cache and dictionary.

entries. In some cases, the reduced custom dictionary overhead is overshadowed by the degraded compression of the binary (e.g., *ghostscript.gs*, 512-entry RT), but most codes actually benefit from dedicating at least some of the dictionary to fixed entries. *Crafty*/2K is a nice example; it achieves the best total compression when only 20% of the dictionary is customized. Most of the codes exhibit a similar valley around 20% or 40%. Although sometimes the best compression is achieved with a completely fixed dictionary (e.g., *g721.dec*), there is usually a significant jump from the 20% bar to the 0% bar, suggesting that some amount of customization is useful.

4.5 Performance Impact

The performance of a system that uses DISE decompression 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 next two sections of the evaluation focus on performance and energy, variations in which are due to trade-offs between the instruction cache and RT.

Instruction Cache Performance. Figure 12 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 15 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 that of a configuration with a 32 KB instruction cache and no (de)compression. Performance, naturally, decreases with cache size, in some cases significantly (e.g., *crafty* suffers five times the number of misses with a 16 KB cache versus a 32 KB cache). Of the three components of average access time—hit time, miss rate, and miss penalty—only the miss rate is impacted by DISE; we fix the miss penalty and ignore the possibility 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 *vortex* show significant improvement when dictionary size is increased beyond 2K instructions.

Counterintuitively, compression may hurt cache performance by producing pathological cache conflicts that did not exist in uncompressed (or less aggressively compressed) code. This effect is more likely to occur at small cache sizes. A prime example is *ghostscript*. Although not immediately evident from the figure, on the 8 KB and 4 KB 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 the utility of programmable compression.

DISE Engine Performance. In contrast with the preceding, here we are concerned with all aspects of RT performance. RT hit time is determined by the DISE engine pipeline organization. The PT and RT are logically accessed in series. In a 2-stage decoder, serial PT/RT access could be hidden with no observed penalty, because both are very small structures. However, adding DISE to a single-cycle decoder requires either an additional pipeline stage that 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. A complete discussion of the performance implications of DISE implementation is available elsewhere [Corliss et al. 2003a].

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

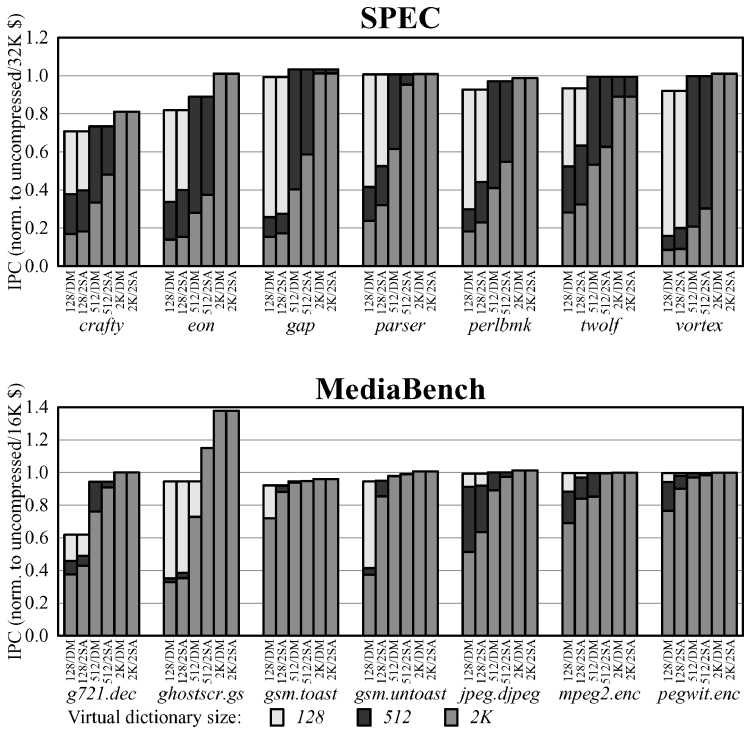

Fig. 13. Performance impact of RT misses.

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 13 shows the performance (i.e., IPC) of systems with several virtual dictionary sizes (128, 512, 2K instructions) on RTs of several different configurations (128, 512, and 2K instruction specification slots arranged in four instruction blocks, both direct mapped and 2-way set-associative). Performance is normalized to the "large instruction cache" (32K or 16K) DISE-free configuration, while the DISE experiments all use smaller caches (16K or 8K) so that the DISE configurations use no more hardware than the baseline. 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 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 they do in SPEC.

4.6 Energy Implications

In a typical general-purpose processor, an instruction cache accesses 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.

Postfetch 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 factors contributing to an increase of energy consumption. 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 perapplication basis by selectively powering down cache ways [Albonesi 1999] or sets [Yang et al. 2002]. A similar strategy can be used for the RT.

Energy and EDP. Figure 14 shows the relative energy consumptions and energy-delay products (EDP, a metric that considers both energy and execution performance) 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 (32 KB or 16 KB) instruction cache, respectively. Each bar shows three energy components: instruction cache (light), DISE structures (medium), and all other resources (dark). Note that 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 performance penalty 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 trade-off between cache and RT instruction capacity highly favors the cache. In the first place, accessing two 16 KB structures consumes more energy than accessing a single 32 KB 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

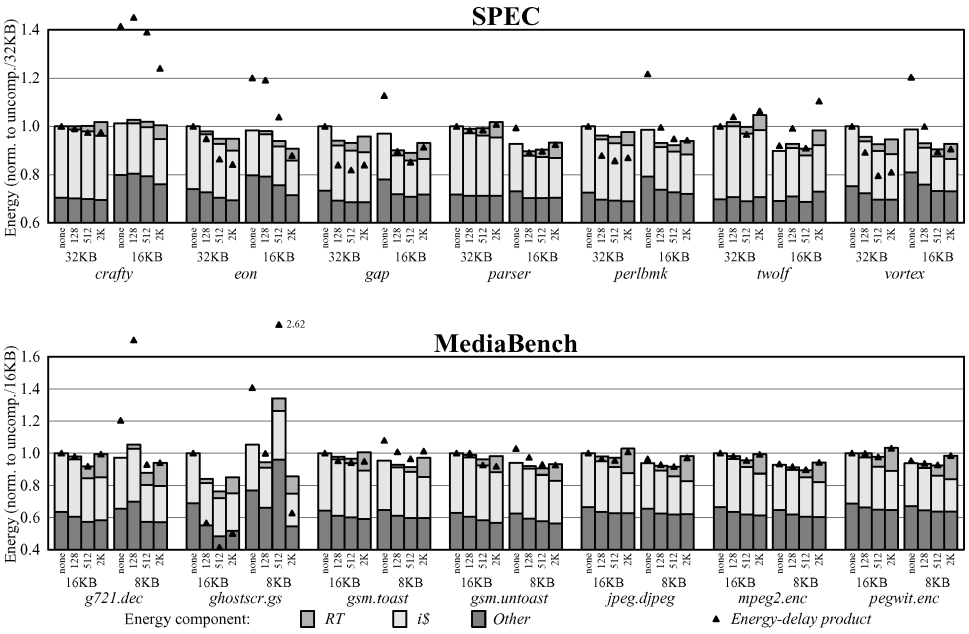

Fig. 14. Impact of compression on energy.

output drivers—is largely independent of array size. Multiple structures also consume more tag power. Our simulations show that a-32 KB single-ported cache consumes only slightly over 40% more energy per access than a single-ported 16 KB cache, not 100% more. Beyond that, however, an RT is less space efficient than an instruction cache because it must store perinstruction instantiation directives as well. When we combine these factors, we see that in order to save energy over a-32 KB configuration, we must replace 16 KB of cache (storage for 4K instructions) with a-3 KB 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 that 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 16 KB 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

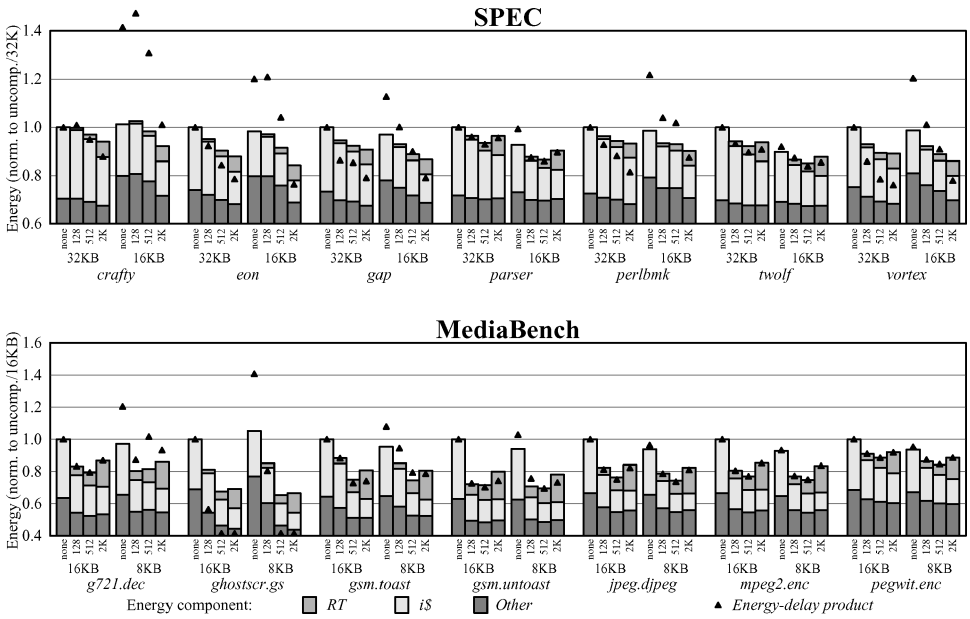

Fig. 15. Impact of profile-based code compression on energy.

number of instruction cache *accesses*. To this point, our compression profiles have been based on static instruction sequence frequency. As a result the statically most frequently occurring sequences are compressed. Alternatively, compression profiles may encode dynamic sequence frequency, allowing us to compress sequences that appear frequently in the dynamic execution stream. Our compression algorithm easily builds compression dictionaries for this scenario. It simply weighs each instruction sequence in a compression profile by an incidence frequency taken from some dynamic execution profile. Although this will likely not reduce code size by as much as the static alternative, compression in this manner will further reduce cache energy.

In Figure 15, we repeat our experiment using dynamic-profile-based dictionaries (note that we have chosen the inputs to these benchmarks to be different from those used in the dynamic profiling step). 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 that the additional EDP reduction comes from the corresponding energy reduction, not from a further reduction in execution time.

4.7 Impact of Compiler and Optimization

Thus far our results have all come from a single compiler performing a large number of execution-time reducing optimizations. In this section we evaluate the impact of compiler and optimization on compressibility. Specifically, we evaluate the impact of various optimization levels (e.g., *−O2* versus *−O3*), various optimizations (e.g., function inlining and loop unrolling), and compilers

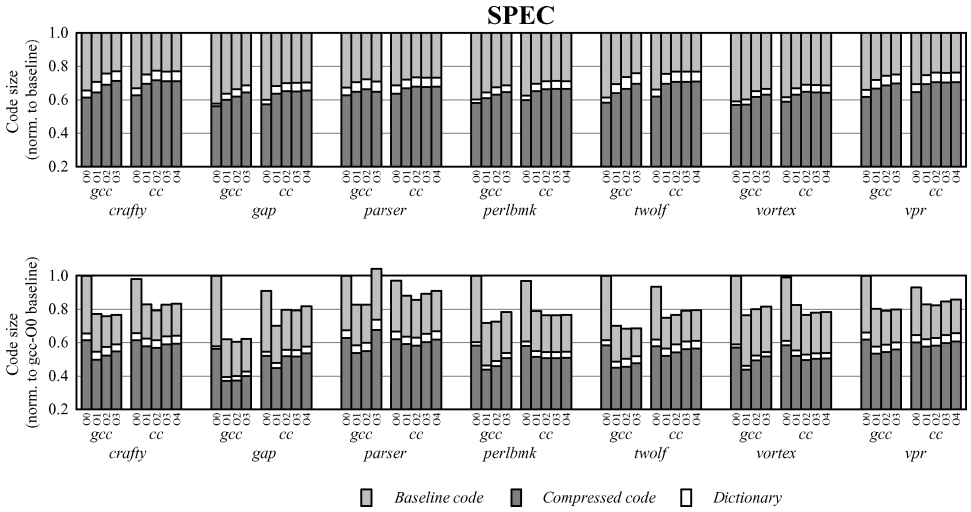

Fig. 16. Impact of compiler and optimization level.

(e.g., GCC versus a native C compiler). We find that the highest optimization levels do not result in the best code compression, so space-critical applications may benefit from less aggressive optimization. For the first experiment we examine only SPEC benchmarks because the MediaBench codes have the same character. We also only examine C compilers, so we replace *eon* (C++) with *vpr*.

Optimization Levels. Figure 16 graphs the impact of optimization level (for GCC 3.3.1, labeled *gcc*, and the native Digital Unix C compiler, labeled *cc*) on compressibility.[1] The top and bottom graphs present the same data, except that all figures in the top graph are normalized to the corresponding uncompressed case (thus all light gray bars reach 1), and in the bottom graph all figures for a particular benchmark are normalized to the uncompressed *gcc-O0* case. The bottom graph allows direct intrabenchmark comparisons (i.e., lower bars imply smaller codes).

From the top graph, it is apparent that lower optimization levels (for both compilers) result in better compression relative to uncompressed code (for that optimization level). The reason for this is that optimization removes regularity from programs. Highly compressible idioms are perturbed by common optimizations such as constant propagation, constant folding, common subexpression elimination, and instruction scheduling. That the dictionaries are often smaller for unoptimized code supports this.

Unoptimized code results in the best relative compression, but it does not necessarily result in the smallest program. Figure 16 (bottom) illustrates this point. Although unoptimized code results in the best relative compression, it dramatically increases the size of the uncompressed executable. For most benchmarks, performing a small amount of optimization (i.e., $-O1$ or $-O2$) results in the best total compression. In some cases, optimization level dramatically impacts

[1]Note that GCC does not have a $-O4$ optimization level.

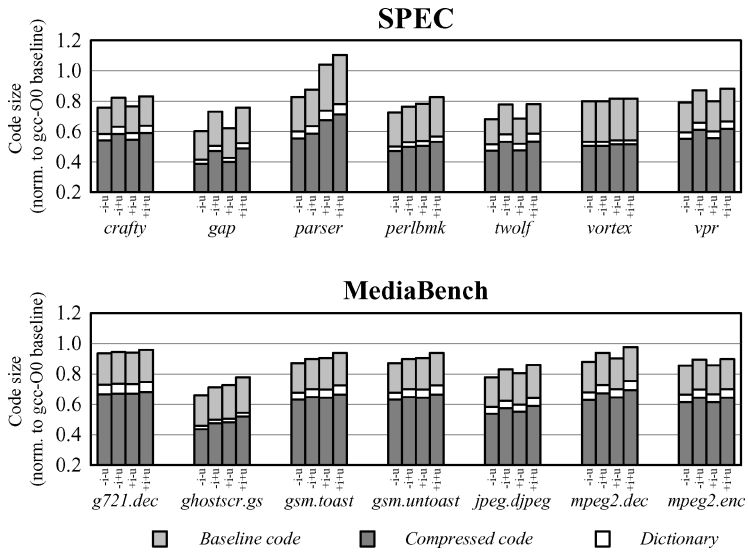

Fig. 17. Impact of function inlining and loop unrolling.

compressibility (e.g., the *parser* benchmark compiled with GCC is more than 25% smaller when compiled with $-O2$ versus $-O3$).

Function Inlining and Loop Unrolling. Next, we assess the impact on compression of particular optimizations. Specifically, we consider function inlining and loop unrolling, because these two transformations can significantly increase code size. In Figure 17, we evaluate the impact of all combinations of performing inlining and unrolling (in the context of GCC's highest optimization level). Each bar is labeled with $+i$ or $-$i indicating whether or not function inlining is performed and $+u$ or $-u$ indicating whether or not unrolling is performed. From this graph, it is clear that most of the benchmarks benefit from some amount of unrolling, because this transformation causes the baseline code size to increase. Conversely, many of the benchmarks (e.g., *crafty*, *gap*, *jpeg*, and *mpeg2*) have little or no opportunity for inlining, because the bars with and without inlining are nearly identical. And some benchmarks (e.g., *parser*, *perlbmk*, and *ghostscript*) benefit from both. Aside from *gap*, for these benchmarks compression is unable to significantly compensate for the code bloat due to these transformations. One might expect better results given that these transformations introduce redundant (and presumably compressible) code, but it appears that other optimizations (e.g., instruction scheduling) are perturbing the compression opportunity.

Optimizing for Both Code Size and Performance. Previously, we saw that lower optimization levels result in better total compression. In most contexts, performance is also an important concern. Figure 18 (top) graphs the execution times of the benchmarks in Figure 16 relative to *gcc-O0*. The light and dark bars are depth sorted to present figures for both the uncompressed baseline and the compressed case, respectively. In one case (*vortex* compiled with *cc* using $-O0$)

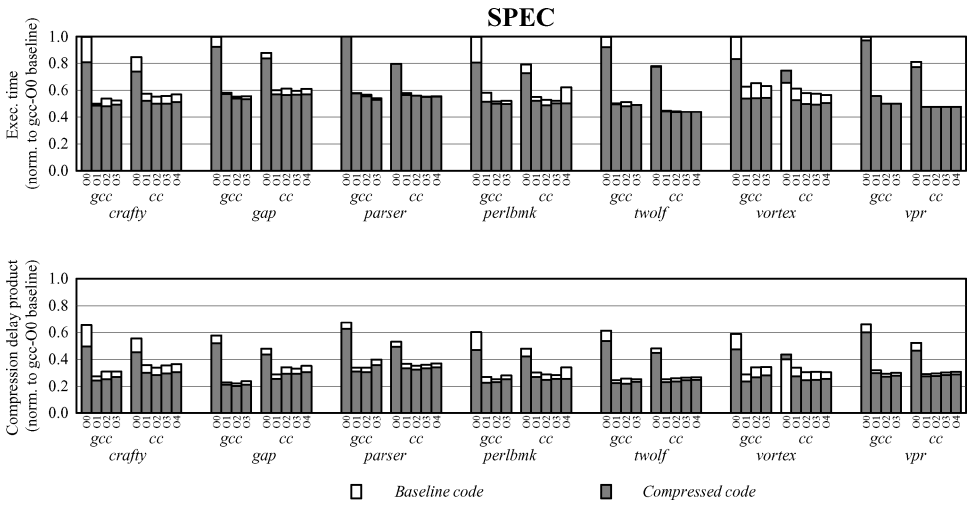

Fig. 18. Impact of optimization on performance and compression delay product (CDP).

the former is superior to the latter, so the baseline (light) bar is placed in front, resulting in the dark portion showing on top. This situation arises when compression degrades performance due to pathological instruction cache conflicts. Generally, *-O0* performance is abysmal, and performance improves with higher optimization levels. In the spirit of the energy delay product (see Section 4.6) we define *compression-delay product* (CDP) analogously, and plot it in Figure 18 (bottom) so that we may optimize both compression and performance. In most cases $-O1$ or $-O2$ results in the best CDP, because the compression is better, yet the loss in performance versus higher optimization levels is minimal.

Compilers. Figures 16 and 18 also illustrate the impact of the compiler on compression. Although the top graph in Figure 16 shows little difference in the relative compression between *gcc* and *cc*, the bottom graph shows that *gcc* usually produces slightly smaller codes resulting in smaller compressed codes. We cannot be certain of the reason for this without reverse engineering the Digital Unix C compiler to learn its optimizations and compare this with GCC. Our hypothesis is that (following the trend that more heavily optimized code is less compressible) the Digital Unix C compiler is a more aggressive compiler customized to the Alpha processor, resulting in less regular and thus less compressible code. Further study is necessary to prove this point.

5. RELATED WORK

The large body of work on code compression speaks to the importance of the technique. In summary, the principal contribution of the present work is the demonstration that a general-purpose dynamic code transformation mechanism (i.e., DISE) can be used to effectively and efficiently implement dynamic code decompression.

Software-Based Approaches. Traditional static optimizations intended to accelerate execution (e.g., dead-code elimination, common sub-expression elimination, register allocation, and so on) often have the side effect of reducing code size [Debray et al. 2000]. We use an already 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 [Cooper and McIntosh 1999; Debray et al. 2000]. ISA extensions have been proposed to reduce this overhead [Liao et al. 1999].

Nonexecutable 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 [Kirovski et al. 1997] and cache-line granularities [Lefurgy et al. 2000]. Although effective in reducing code size, the performance of these systems degrades significantly. An interesting extension to these works builds on the observation that decompressing frequently executed code slows execution; therefore by compressing only infrequently executed code, code size is reduced yet execution time is minimally degraded (say, 4% [Debray and Evans 2002]). In contrast, hardware post-fetch decompression implementations like DISE can actually reduce execution time and energy by specifically concentrating on (de)compressing frequently executed code.

Ernst et al. [1997] describe a compressed code format (byte-code RISC or BRISC) that may be directly interpreted (i.e., it does not require a separate decompression step). The representation includes a form of operand parameterization. Although highly effective in an interpreted environment, the approach is too expensive for hardware implementation. Furthermore, the results rely on the particular BRISC representation, while we show results for multiple ISAs.

ISA Extensions. Certain ISAs (e.g., ARM's Thumb [Advanced RISC Machines Ltd. 1995] and MIPS16 [Kissell 1997]) 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). A recent ISA extension, Thumb-2, better balances the compression/performance trade-off, approaching the compression levels of Thumb without as significant a performance degradation [Phelan 2003]. Nevertheless, dense instruction encodings do not exploit repetition of code sequences like coarse-grained (i.e., multiple instruction) (de)compression schemes. Dense encodings and coarse-grained (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) [Wolfe and Chanin 1992] and IBM's CodePack [Kemp et al. 1998]. Fill-path decompression schemes necessitate no

processor core modifications and incur decompression cost only on instruction cache misses. Although in rare cases they may improve performance (e.g., CodePack implements a form of prefetching), 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) are compressed rather than instruction sequences.

DISE performs postfetch 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 postfetch dictionary decompression using custom hardware has been previously proposed. One such system [Lefurgy et al. 1997] 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 postfetch decompression system [Lekatsas et al. 2000] 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 [Araujo et al. 1998] extends postfetch 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.

Nam et al. [1999] propose a VLIW postfetch decompression system that supports a variant of operand factorization. Individual VLIW instruction words are compressed to indices in opcode and operand dictionaries. The number of instructions encoded by a single compressed instruction is a function of the number of operations that appear in each VLIW instruction (i.e., longer or shorter encodings are not possible). Nam et al. also find that compressing similar but not identical sequences (via *instruction isomorphism*) dramatically improves compression effectiveness.

6. CONCLUSION

Code (de)compression is an important tool for architects of both embedded and general-purpose microprocessors. In this paper, we present and evaluate 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 [Corliss et al. 2002, 2003a]. A DISE implementation of (de)compression has many advantages. It implements postfetch 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 exist 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.

This work includes an extensive experimental evaluation in which we not only measure code compression itself, but also evaluate its impact on dynamic characteristics such as performance and energy. We show that DISE enables code size reductions of 25% to 35% and often results in better compression than previously proposed custom compression hardware. We measure the impact of DISE-specific features or attributes, such as parameterization, branch compression, and demand-loading the dictionary. We find that the most unique DISE feature (versus other hardware approaches to dynamic decompression), parameterization, dramatically improves its ability compress (by up to 20%) and allows PC-relative branches to be compressed. We also evaluate a number of issues that have implications for any postfetch decompression mechanism. For example, we find that application-customized dictionaries enable better compression than fixed dictionaries, and a hybrid of the two is still better. We find that very large dictionaries are unnecessary and that different compilers using different optimizations produce code of different degrees of compressibility. We also quantify the impact of compression on performance (in some case a 20% improvement) and energy (in some cases a 10% reduction).

ACKNOWLEDGMENTS

The authors thank Vlad Petric for his help with the energy simulations and the anonymous reviewers for their insightful comments.

REFERENCES

ADVANCED RISC MACHINES LTD. 1995. *An Introduction to Thumb*. Advanced RISC Machines Ltd, Austin, TX.

ALBONESI, D. 1999. Selective cache ways: On demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*. 248–259.

ARAUJO, G., CENTODUCATTE, P., AND CORTES, M. 1998. Code compression based on operand factorization. In *Proceedings of the 31st International Symposium on Microarchitecture*. 194–201.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*. 83–94.

BURGER, D. AND AUSTIN, T. M. 1997. *The SimpleScalar Tool Set, Version 2.0*. Tech. Rep. 1342, University of Wisconsin–Madison Computer Sciences Department.

COOPER, K. AND MCINTOSH, N. 1999. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. 139–149.

CORLISS, M. L., LEWIS, E. C., AND ROTH, A. 2002. *DISE: Dynamic Instruction Stream Editing*. Tech. Rep. MS-CIS-02-24, University of Pennsylvania. July.

CORLISS, M. L., LEWIS, E. C., AND ROTH, A. 2003a. DISE: A programmable macro engine for customizing applications. In *Proceedings of the 30th International Symposium on Computer Architecture*. 362–373.

CORLISS, M. L., LEWIS, E. C., AND ROTH, A. 2003b. A DISE implementation of dynamic code decompression. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. 232–243.

CORMIE, D. 2002. The ARM11 microarchitecture. ARM Ltd. White Paper.

DEBRAY, S. AND EVANS, W. 2002. Profile-guided code compression. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation*. 95–105.

DEBRAY, S. K., EVANS, W., MUTH, R., AND B. DE SUTTER. 2000. Compiler techniques for code compression. *ACM Trans. Program. Lang. Operating Syst. 22*, 2 (Mar.), 378–415.

DIEFENDORF, K. 1998. K7 challenges Intel. *Microprocess. Rep. 12*, 14 (Nov.).

ERNST, J., EVANS, W., FRASER, C., LUCCO, S., AND PROEBSTING, T. 1997. Code compression. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. 358–365.

GLASKOWSKY, P. 2000. Pentium 4 (partially) previewed. *Microprocess. Rep. 14*, 8 (Aug.).

GWENAPP, L. 1997. P6 microcode can be patched. *Microprocess. Rep. 11*, 12 (Sep.).

KEMP, T. M., MONTOYE, R. K., AUERBACK, D. J., HARPER, J. D., AND PALMER, J. D. 1998. A decompression core for PowerPC. *IBM Syst. J. 42*, 6 (November), 807–812.

KIROVSKI, D., KIN, J., AND MANGIONE-SMITH, W. 1997. Procedure based program compression. In *Proceedings of the 30th International Symposium on Microarchitecture*. 204–213.

KISSELL, K. 1997. *MIPS16: High-Density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, Mt. View, CA.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings 30th International Symposium on Microarchitecture*. 330–335.

LEFURGY, C., BIRD, P., CHENG, I.-C., AND MUDGE, T. 1997. Improving code density using compression techniques. In *Proceedings of the 30th International Symposium on Microarchitecture*. 194–203.

LEFURGY, C., PICCININNI, E., AND MUDGE, T. 2000. Reducing code size with run-time decompression. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*. 218–227.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000. Code compression for low power embedded system design. In *Proceedings 36th Design Automation Conference*. 294–299.

LIAO, S., DEVADAS, S., AND KEUTZER, K. 1999. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Design Autom. Electr. Syst. 4*, 1 (Jan.), 12–38.

NAM, S.-J., PARK, I.-C., AND KYUNG, C.-M. 1999. Improving dictionary-based code compression in VLIW architectures. *IEICE Trans. Fundam. E82-A*, 11 (Nov.), 2318–2324.

PHELAN, R. 2003. *Improving ARM Code Density and Performance*. Tech. Rep., Advanced RISC Machines Ltd, Austin, TX.

SZYMANSKI, T. 1978. Assembling code for machines with span dependent instructions. *Commun. ACM 21*, 4 (Apr.), 300–308.

WILTON, S. AND JOUPPI, N. 1994. *An Enhanced Access and Cycle Time Model for On-Chip Caches*. Tech. Rep., DEC Western Research Laboratory, Palo Alto, CA.

WOLFE, A. AND CHANIN, A. 1992. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the 25th International Symposium on Microarchitecture*. 81–91.

YANG, S.-H., POWELL, M., FALSAFI, B., AND VIJAYKUMAR, T. 2002. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings 8th International Symposium on High Performance Computer Architecture*.

Received October 2003; revised April 2004; accepted July 2004