Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors

Sergei Y. Larin Thomas M. Conte

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7914
conte@ncsu.edu

Abstract

During the last 15 years, embedded systems have grown in complexity and performance to rival desktop systems. The architectures of these systems present unique challenges to processor microarchitecture, including instruction encoding and instruction fetch processes. This paper presents new techniques for reducing embedded system code size without reducing functionality. This approach is to extract the pipeline decoder logic for an embedded VLIW processor in software at system development time. The code size reduction is achieved by Huffman compressing or tailor encoding the ISA of the original program. Some interesting results were found. In particular, the degree of compression for the ROM doesn't translate into an improvement in instructions delivered per cycle. Experiments found that when the missprediction penalty of the added Huffman decoder stage was taken into account, a Tailored ISA approach produced higher performance. Methods that compress the entire operation using Huffman encodings, and decompress at ICache hit time still achieved a median performance advantage, while providing higher ROM size savings. All results were generated by an optimizing compiler and tool suite, and presented for an encoding similar to the Intel/HP IA-64 architecture.

1 Introduction

The importance of embedded systems (ES) today is easy to underestimate. During the last 15 years ESes have grown rapidly in complexity and performance to a point where they now rival the design challenges of desktop systems. For example, ESes have a set of contradictory requirements: ESes are expected to occupy small physical space (e.g., low package count), be inexpensive, highly reliable-- and yet they are asked to take on more and more complex functions. The specific architectures of these systems present unique challenges to processor microarchitecture, including instruction encoding and the instruction fetch processes. This paper presents new techniques for reducing embedded system size and, indirectly, power consumption, without reducing functionality.

A standard approach to building an ES is by using an ASIC. On such a system, all code is stored in a ROM and an on-chip commercial core processor is used (see Figure 1). One problem with this approach is that, with the growth in device functionality, the ROM size multiplies. Soon it becomes the major cost defining factor and bottleneck for instruction fetch (IFetch) [1,3,14,27,28]. In addition, the code ROM is often implemented on a separate chip, which involves familiar difficulties associated with remote IFetch and off-die power consumption. The challenge is to reduce the size of the ROM without sacrificing functionality and performance of the system. Several prior approaches to this problem have either defined new instruction-set architectures (e.g., the ARM Thumb [25] or SGI MIPS16 [26]) or defined compression schemes without taking the impact on IFetch into account (e.g., IBM CodePack [9], Cooper and McIntosh [11]). (One notable exception involves inferring subroutines from normal code sequences [14].) In contrast, this paper takes a unified, compiler-driven approach to the problem. It presents both code compression schemes and their corresponding instruction fetch mechanisms. All results are generated using an optimizing compiler built for the task by the authors.

With the growth of ES application complexity far beyond familiar embedded DSP applications, the traditional ways of hand coding and optimizing have become less and less effective and increasingly time consuming. The market requirements for a short design cycle become a limiting factor. A practical way to use a high-level programming language while maintaining optimal target code quality is needed [3]. For example, the TI 320C6xxx series of DSP processors are successful in large part due to the vendor-supplied optimizing compiler. This paper follows this trend by focusing on compiler-driven code design for ES applications. Since an ES is likely to execute a code base throughout its life span, the compiler can generate an efficient *custom-tailored instruction set architecture (ISA)* in addition to optimizing the code. In this study the ISA is optimized for space and cache performance, but it could also be improved for power consumption, branch/data prediction accuracy, or other purposes.

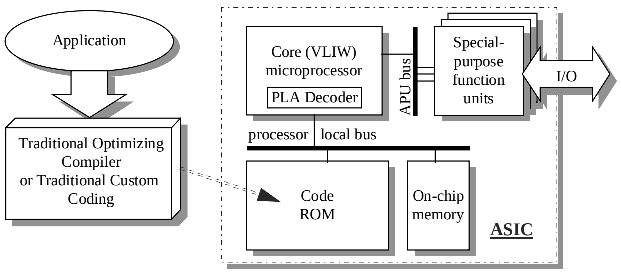
Application
Traditional Optimizing Compiler or Traditional Custom Coding
Core (VLIW) microprocessor
PLA Decoder
APU bus
Special-purpose function units
I/O
processor local bus
Code ROM
On-chip memory
ASIC

Figure 1. Traditional ASIC Embedded System organization.

This paper introduces a novel, unified approach to solving code size issue *together* with the IFetch mechanism. We use the natural flexibility of the ASIC approach to reduce the ROM size significantly (e.g., a 20% to 75% size reduction) without impacting the performance of the system. On the contrary, some *increases* (5-10%) in performance have been achieved compared to an uncompressed implementation. We also reduce the memory bus usage, which promise savings in power consumption.

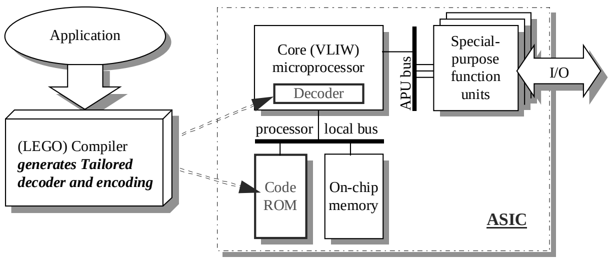
Application
(LEGO) Compiler
generates Tailored decoder and encoding
Core (VLIW) microprocessor
Decoder
APU bus
Special-purpose function units
I/O
processor local bus
Code ROM
On-chip memory
ASIC

Figure 2. Proposed approach to Embedded System design.

Another element of this work involves the use of VLIW architectures for ESes, similar to the commercial TI DSP processors [27]. In general, VLIW seems to be a natural fit since many of the traditional VLIW problems are not applicable to ESes. There is almost no code compatibility problem between generations. Simpler hardware leads to higher performance and lower power consumption when compared to an equivalent issue-width superscalar architecture. Since extensive information is available about the dynamic behavior of the program, high quality schedules can be produced by the compiler. The challenges are in static code size and effective IFetch. The object code size difference between a VLIW and Superscalar is reduced by using the zero-NOP encoding [7] and by restricting code duplication in the compiler to RISC-like levels. IFetch issues have been discussed [7,8] elsewhere and are extended in this paper.

What makes the approach in this paper possible is the fact that many ESes use a PLA as a decoder for the core processor, which in turn can be *reprogrammed* to decode a custom ISA. The decoder's structure is produced by the compiler, which has all the information needed. The overall structure of such a system is presented in Figure 2. Thus the compiler plays a role in dictating not only the ROM contents, but also the core processor decoder logic.

The rest of this paper is organized as follows. Section 2 describes code compression. In Section 3, the IFetch tradeoffs are analyzed. Sections 4 and 5 propose specific ways to implement an ICache. Section 6 discusses previous studies and their relation to this paper, and Section 7 concludes the work.

2 Tailored Encoding and Compression techniques

Traditionally, there are two general approaches to reduction of program size. One is the reduction of the number of assembly operations in the code [8,11,12,14,26], and the other is the reduction of the operation size [1,9,27,28]. Generally, these two approaches both try to increase the entropy of the code, but in different ways. This means that applying *both* of them to the same code often will not result in better compression. This paper concentrates on reducing the operation size. A separate study will consider VLIW ES IFetch mechanisms for operation reduction techniques.

There are two possibilities to reduce operation size: *tailor the ISA* or *compress the code*. The tailored ISA is a new encoding which is generated for one particular program/application, and best fits it's characteristics. After decoding of a tailored operation, the internal processor signals are obtained. The compression of code takes an existing encoding, and, according to the static frequencies of elements in the source code, determines the best way to pack it. Theoretically the former should yield better performance (e.g., no intermediate decompression needed), while the latter yields a smaller code size. This paper finds that this intuition holds even when the improved instruction cache performance from caching compressed code is taken into account.

2.1 Target Architecture Description

Our experiments are based on the TEPIC (TINKER EPIC) Embedded architecture [10]. It is a 40 bit version of the HP PlayDoh VLIW machine specification [20] adapted for ES. (It is interesting to note that this encoding is very similar to the recently announced Intel/HP IA-64 ISA.) For the core processor, we assume a 6 issue machine with 4 units that can execute any instruction except for memory access and 2 universal units (including memory accesses). The register files are fixed to 32 GPRs, 32 FPRs and 32 predicate registers. The encoding formats for operations are shown in the Appendix. This encoding is known as Zero-nop [7] encoding and was designed to hide code expansion associated with VLIWs. RISC-like ops are combined into VLIW multiops (MOPs) during instruction scheduling. By use of some additional fields (i.e., the tail bit) TEPIC does not have to store NOPs. We use the LEGO optimizing compiler [6], which employs standard optimizations and global instruction scheduling using Treegions [4,5,6] to schedule and optimize the code. A modified version of the TINKER assembler is used to generate custom encodings as well as the synthesizeable Verilog for the decoders (where applicable, e.g., for tailored ISAs). The output is runnable code that is emulated via the TINKER YULA emulation tool. Annotations are added by the compiler to emit an instruction address trace for cache simulations (these annotations are not included when determining instruction addresses or performing compression).

2.2 Compression techniques

In general, compression circumstances for embedded systems are favorable, since the entire code image is available statically for the compression algorithm. A fast PLA-based hardware decoder is used for decompression. The main issue is the complexity, and therefore size, of the decoder. There are several potential compression algorithms to choose from. For this environment, Huffman algorithm [2] produces near optimal results. It also allows fast decompression at reasonable real estate price [17,18]. For similar reasons, it was used in several previous studies [1,9].

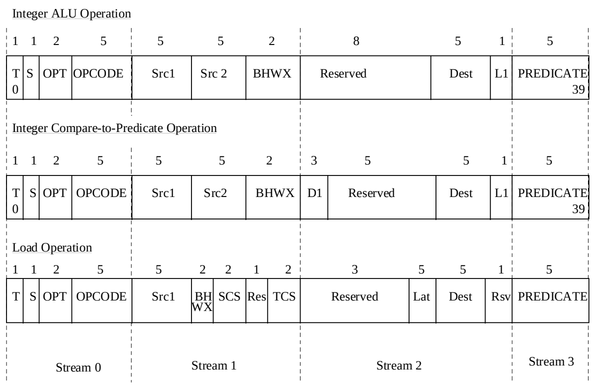
Integer ALU Operation
1 1 2 5 5 5 2 8 5 1 5
T0 S OPT OPCODE Src1 Src 2 BHWX Reserved Dest L1 PREDICATE 39
Integer Compare-to-Predicate Operation
1 1 2 5 5 5 2 3 5 5 1 5
T0 S OPT OPCODE Src1 Src2 BHWX D1 Reserved Dest L1 PREDICATE 39
Load Operation
1 1 2 5 5 2 2 1 2 3 5 5 1 5
T S OPT OPCODE Src1 BHWX SCS Res TCS Reserved Lat Dest Rsv PREDICATE
Stream 0
Stream 1
Stream 2
Stream 3

Figure 3. Stream Based Huffman Alphabet.

We consider three different ways to compose the input alphabet for the Huffman algorithm. First is the traditional *byte*-based method [1,2]. The code segment is considered as a stream of bytes and compressed accordingly. As we will see shortly this method produces the smallest decoding table and the simplest decoder. Second is the *stream*-based approach. The idea behind this approach is that certain fields in an instruction encoding exhibit more repetitive patterns when taken as independent compression streams than when combined with other fields (see Figure 3 and Table 2 in Appendix). For example the OpType/OpCode fields of the TEPIC instruction are set to 'INT_OpType and ADD_OpCode' very often. The same is true for the predicate field, which is most of the time is set to 'true'. According to this observation, alphabet streams are fixed at certain operation field boundaries, as shown in Figure 3.

The last approach to alphabet composition is to consider the *whole Op* as a compression unit. This method produces the largest decoding table but has the greatest potential for compression. This result becomes more understandable when one examines the generated Huffman codes. Even with a large number of dictionary entries, the size of the popular ADD instruction often went down from 40 to 6 bits, and none of the codes exceed the original op size. In contrast, the maximum degree of compression of either *stream* approach is the sum of the maximum compression of all four streams, which easily exceeds 6 bits in most cases.

The important observation is that combining two or more compression strategies does not yield better compression, since we are approaching *entropy* limit of the program. One additional detail of Huffman compression involves symbol length. For some inputs, Huffman will produce very long output codes that are incompatible with IFetch hardware. The compiler keeps track of such events and either alternates the compression process (similar to the Bounded Huffman code described by Wolfe [1], where some additional encoding is used) or substitutes the rare instruction with an equivalent group of more common ones (e.g. strength reduction).

2.3 Tailored Encoding

The idea behind Tailored encoding is to give the op as much space as it needs but not to compress it otherwise. This method still gets significant space savings compared to the original ISA, but avoids decompression entirely. As the tailor-encoded instruction is decoded the core processor's internal signals are obtained directly. In this study, the Verilog code for the decoder is produced by the compiler and used to configure the PLA.

Tailored encoding generation is straightforward. If the program uses less than eight floating-point operations, the FP OpCode field only needs three bits. Similarly, after register allocation, if no more than four

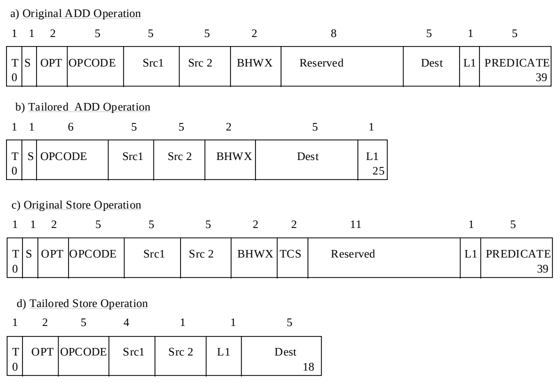
a) Original ADD Operation
1 1 2 5 5 5 2 8 5 1 5
T 0 | S | OPT | OPCODE | Src1 | Src 2 | BHWX | Reserved | Dest | L1 | PREDICATE 39
b) Tailored ADD Operation
1 1 6 5 5 2 5 1
T 0 | S | OPCODE | Src1 | Src 2 | BHWX | Dest | L1 25
c) Original Store Operation
1 1 2 5 5 5 2 2 11 1 5
T 0 | S | OPT | OPCODE | Src1 | Src 2 | BHWX | TCS | Reserved | L1 | PREDICATE 39
d) Tailored Store Operation
1 2 5 4 1 1 5
T 0 | OPT | OPCODE | Src1 | Src 2 | L1 | Dest 18

Figure 4. Tailored Encoding Example

registers of some type are live at the same time in some source position, it needs only two bits, etc. The result is an *uncompressed*, but *compact* version of the original program (see Figure 4). While forming a tailored ISA, some enhancement is possible for decoding. For instance, if every instruction has its Tail bit, OpType and OpCode fields in a fixed position (and possibly of a fixed size), it significantly simplifies decoding (no search needed). The compiler looks for opportunities like this when it generates the tailored ISA.

The comparison between all methods is presented in Figure 5 for the code segment only (see Section 3.3 below). Six stream configurations where considered, and the best two are shown in the Figure 5.

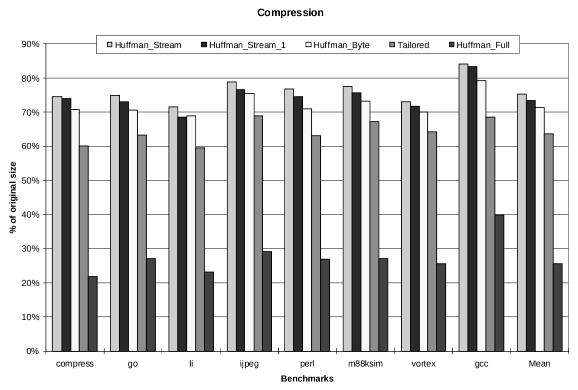

Figure 5. Different Compression Techniques comparison (code segment only).

In general, the choice of best possible stream encoding is an exponential time task. The encodings presented here were selected for the smallest code size (stream_1) and for the smallest decoder (stream) among considered variations. Also presented is byte-wise compression, which views the image as a byte stream with no other intelligence. This is in contrast to Full, which compresses the image on an operation by operation level (40b each). Note the remarkable code size reduction with the Full compression scheme (30% of original size on average). But, as we will see in Section 3.4, it produces a very large decoder, which in turn might prevent the use of it as the primary compression algorithm. The tailored ISA approach produces code on the order of 64% of the original size, which can have favorable results for very little additional hardware overhead. Both of these results neglect the address target table, which is discussed in Section 3.3 below. It adds approximately 15.5% to the image size. The impact of the IFetch mechanisms for both schemes are considered below.

3 Instruction Fetch Organization

A significant component of this paper is the joint consideration of encoding and IFetch organization. The fact that the ICache holds *compressed* instructions increases its capacity and, as result, the overall throughput. This leads us to a paradoxical conclusion that we can improve overall performance of the system by applying a compression technique even though more work must be done to interpret the code. The down side of this is that cache control needs to be designed differently to handle compressed instructions. On the positive side, the cache data path design is not dependent on any particular encoding. This makes modular core processor design possible.

3.1 Program Layout

Let us define *a block* as a sequence of instructions guaranteed (or likely to be) executed sequentially once we start execution of the first instruction in the block. The simplest example of a block is the *Basic Block* (single entry and single exit point). More sophisticated examples are a sequence of basic blocks with no side entrances (like *Superblocks* [21] or Fisher-style *Traces* [15]). Let us consider the simplest type for now, the basic block (BB). A BB can be treated as an *atomic unit of instruction fetch* (see Figure 6).

This implies that the cache can be accessed initially for the first op in the BB. After this, the cache can then supply ops in a streaming fashion until the end of the BB is reached. Then the next BB is predicted. This is a valid approach for the following reasons. First, control transfer can only occur to the first op of a BB. Second, a BB should always be executed from the beginning to the end unless an interrupt has occurred, and even then its execution will be completed after the interrupt has been handled (here subroutine calls are considered to be branches that end a BB). All the necessary NextPC computations local to the block are done within the cache and are hidden for the processor core as long as correct VLIW group is forwarded to the decoder every cycle.

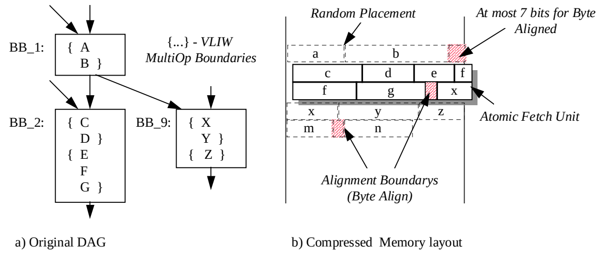

Figure 6. Atomic Block structure

Use of more complicated blocks is a matter of performance, not correctness. If the block is allowed to have side exits, we should guarantee that they are not taken frequently (or we will over-pollute the ICache). This is true for superblocks and Fisher-style traces, which are formed at compilation with the use of profile information. But it is also true that for complex blocks some additional invalidation mechanism is needed. However, in this study we only consider BBs. (Note however that the code is scheduled by first building trees of basic blocks (i.e., treegions), then decomposed into basic blocks after scheduling.)

3.3 Address space considerations

A critical issue for execution of any compressed program is the change in branch target addresses. Some kind of translation must be done. One solution is to convert the original branch targets to the compressed ones at compilation. It could be done in two passes: first, new code layout and new target addresses are generated (with enough space left for later 'plug in' of new targets in relative branches), then on the second pass, new addresses are inserted into the target slots and jump tables are updated. This method better fits tailored ISAs than compressed ones because compressed code with new targets will have to be recompressed with certain restrictions (or branch instructions should remain uncompressed) while the tailored code stays uncompressed naturally.

Another solution to the branch target problem is to leave the original target addresses the way they are (just compress them along with the rest of the code) and provide a dynamic translation mechanism at run time. This approach is very well known in general purpose computing for mapping of virtual address space to the physical one via the TLB. Similar hardware, named Cache Lookaside Buffer (CLB), is also employed in studies by Wolfe, et al. [1,16] and proven to be effective. We use a similar approach to map the original address space to the compressed one with compiler aid. The hardware is called the *Address Translation Buffer* (ATB) and the static table is the *Address Translation Table* (ATT). The ATB holds pairs of addresses, which maps the original address space to the compressed one along with information to aid decoding and Next PC computation. The ATT has *one entry* per each block. It is generated by the compiler and stored in code memory in compressed form. Portions of it are uploaded to the ATB as needed. Due to the normally high spatial locality, the ATB has a very low level of contention and the ATT has small static size (see Figure 7). In general, the ATT adds approximately 15.5% to the size of the ROM.

The additional information in the ATB includes the number of memory lines that need to be fetched in order to get the whole block and the number of ops in the block (or simply the number of VLIW groups in the block). (For simplicity, we will refer to VLIW groups as MultiOps or MOPs for the remainder of the paper). Each MOP is encoded as a collection of ops. This is done by using a dedicated *tail bit* in every op, which is set only for the last op in a MOP [7]. At run time the ATB provides the following information: the address of the requested block in compressed memory, the PC of the last op in the block, and the *predicted* PC of the following block. This is enough to fetch blocks in pipelined fashion.

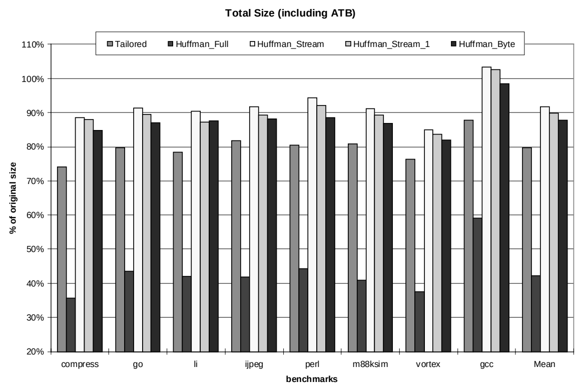

Figure 7. ATB Characteristics. Total code Size.

The next issue in this category is ROM access. The vast majority of modern memory systems support only *byte* or *word* aligned accesses. This puts some, but not critical, limitations on code placement in a compressed stream. We address this by aligning *the first op* of a block to *byte* boundaries. All consecutive ops are sequentially placed in memory.

3.4 Instruction cache design

The instruction cache is a critical element of any high performance system and especially important for this study. The main ICache tradeoff in a compressed system is what address space it belongs to. In other words, whether it holds compressed or uncompressed ops. Most of the researchers [1,8,9] uncompress their instructions prior to putting them into the ICache but a compressed cache is able to hold several times more instructions than an uncompressed one. The only problem is that some work must be performed at the hit path, which potentially increases the branch misprediction penalty (if the cache is pipelined) or stretches the cycle time. However, since the height of the cache can be reduced now with no loss in performance, cycle time stretch is less likely.

The next issue is the NextPC calculation. A cache that supports a zero NOP encoding employs a NextPC calculation mechanism [7,8] that is applicable to this study as well. Let us differentiate the NextPC *within a block* and the NextPC of the *next block*. The NextPC within a block does not need to be predicted since by definition we are going to fetch the block till the end, but rather can be locally calculated with dedication of some additional hardware [8]. This hardware (along with access pattern) varies with *placement* and *invalidation* policy. If a block is *atomically* (sequentially) placed in the cache, intermediate instruction accesses do not have to be checked for validity (we only check against the provided LastPC to determine when the block ends). This will be called the *restricted* placement model.

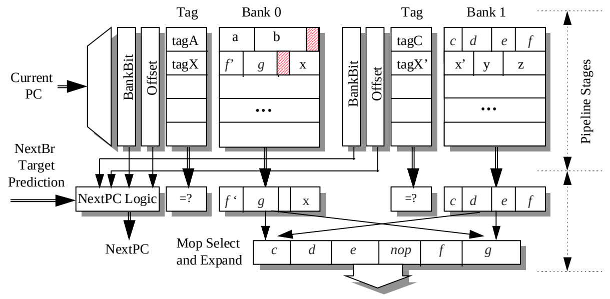

Figure 8. Base line Banked Cache Structure

In a VLIW processor, all ops in a MOP must be issued together. If pieces of the block are allowed to be scattered around the storage, we need to generate an intermediate PC for each MOP, and locally 'reaccess' the cache to check if we have the valid data present. In the latter case, all additional information (individual MOP length, for instance) could be extracted from the fetched block during miss repair time. In the tailored ISA approach, this is especially easy to do since the size of all ops of the same *type* and *code* is the same, and location of this information are fixed within an op (see Section 2.3). For the compressed encoding approach, this information might be generated by the compiler and stored with the ATT in compressed form. Nevertheless, in the current study, we use the *restricted* placement model. The Next PC of the *next block* is the more traditional branch target problem. It needs to be dynamically predicted if we want to achieve full capacity for the IFetch pipeline. This prediction is more important if sophisticated IFetch is used. In the current study, we coupled the branch prediction table with the ATB. This means that for every block entry, there is one branch predictor with *taken/not-taken* and *target address* prediction information. It predicts the outcome of the last instruction of the block (which by definition is often a branch). To predict the outcome of the branch, a two-bit saturating counter is used [13]. To predict the target address, the 'last-target address' (if branch predicted taken), or next sequential address (otherwise) predictor is used. Theoretically more complex branch predictors could be used (e.g., gshare or PAs Yeh/Patt predictor) since likely there will be several cycles to access the prediction unless the code has multiple sequential branches with no other computation in between.

The baseline cache that was selected for this study is the *Banked Cache* we described in [7,8]. Originally designed to fetch variable length MOPs, it fits all the requirements outlined earlier in this section. The structure of it is depicted in Figure 8. The storage of the cache is separated into two banks, similar to the one in the Intel Pentium processor [19]. The bank line size is equal to the maximum size MOP, which in turn is proportional to the issue width of the processor core. A MOP can begin at arbitrary locations and span two cache blocks but still can be extracted in one reference to the bank storage. Two blocks are brought down to the *alignment* stage on cache hit-- the block that was referenced by the PC, and the next sequential block. If the beginning of a MOP resides in Bank1 at index *N*, the next sequential block is taken from Bank0, index *N*+1. The MOP is guaranteed to be within these two cache blocks. Then the alignment hardware scans ops for Tail bits (which mark boundaries of a MOP), and extracts the MOP. NextPC is locally computed in parallel with the alignment stage. More details on the baseline Banked Cache design can be found in [7,8].

3.5 Decoding complexity

Since, in the case of code compression, we choose to place decoding on the critical path of the IFetch mechanism, it should be made as efficient as possible.

In essence it is now a critical factor for the compression algorithm's selection. As we mentioned before, the compiler generates a Verilog description of the decoder that is used to program the PLA (after possible optimization). In the case of the Tailored encoding, this problem is somewhat less critical. Decoding of tailored instructions is a part of the processor pipeline. Nevertheless, compared to the traditional fixed-size-op decoder, it is more complex.

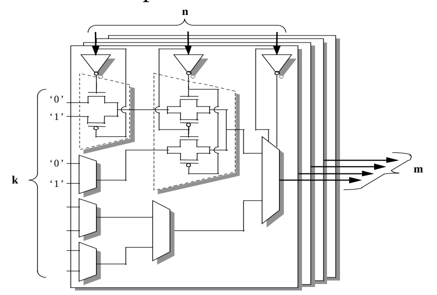

Figure 9. The Huffman Tree decoder structure

As a method of comparison of the Huffman decoder size (a simplified cost function), we can evaluate the complexity of a corresponding *Huffman tree*. If we imagine the structure presented in Figure 9 (where *n* is the longest Huffman code size, *k* is the number of entries in the Huffman dictionary and *m* is the longest dictionary entry size), it is possible to derive an equation to estimate the *worst case* decoder complexity. It is not intended to suggest real hardware implementation, only as a criterion for evaluation. The worst case number of elements (transistors) in a Huffman decoder can be expressed as follows:

$$T = 2m(2^n - 1) + 4m(2^n - 2^{n-1} - 1) + 2n$$

This equation assumes multiplexer implementation using CMOS transmission gates (TG) (two transistors per multiplexer) and accounts for the fact that the first row passes constants and needs only one transistor to operate. Elements to form inverters are included as well. Assuming this model, we can evaluate complexity of the various Huffman decoders (see Figure 10). This Figure in conjunction with Table 1 allows us clearly to see the tradeoff between decoder complexity and degree of compression. The best compression algorithm (Huffman Full) yields the largest decoder size. This relationship is not necessarily linear. Byte-wise compression yields an intermediate degree of code size (72% of the original image size) yet has the smallest decoder (See Figure 5). The worst Huffman compression scheme, Stream, achieves approximately 75% of the original image size, yet has a significant decoder complexity. The reason is the limited input width and dictionary size of Byte-wise compression.

Practical implementations of the Huffman decoder in hardware have been proposed in two studies [17,18]. Both are strongly dependent on implementation (MPEG-2 decompression for example), but generally can achieve 300-600 Mbit/sec for a table with 114 dictionary entries and codes in the range of 1 to 16 bits. The real-estate budget ranges from 10,000 to 28,000

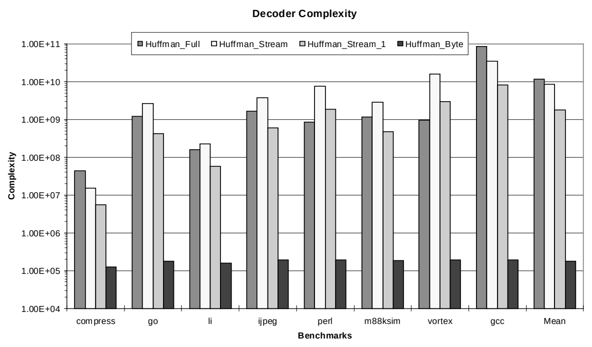

Figure 10. Huffman Decoder Complexity.

transistors. This data allows an assumption for the time needed to decompress the code. For the 20-50ns cycle times typical in embedded processors, we can assume that a decoding of 40 bits (the op size in the baseline TEPIC architecture) is practical. Therefore, it is assumed that one op could be decoded in a cycle (or we can say that the time to decode one op defines the cycle time). Note that all these assumptions are only relevant for Compressed encodings, and do not apply to Tailored encoding.

4 The ICache design for Compressed Encoding

The implementation of the ICache for compression is designed to lessen the impact of decompression time on the IFetch rate. One block is decompressed at a time and is held in a *buffer*, which is accessed in parallel with (but has priority over) the main cache. This buffer is organized as a small fully associative cache. In general, the whole structure could be seen as two-level ICache, where decoding is done at miss time of the L1 cache and the buffer is in essence an "L0" cache. The main cache storage is organized as the Banked Cache described in Section 3.4. It has double bank storage and an alignment network to guarantee fetch of a whole MOP in pipelined fashion. Coupled with the ATB, it is accessed only for the first op in a BB and keeps supplying MOPs until the end of the block when the next block's address is predicted. The size of the L0 buffer was set at 32 op entries (160 bytes). From our experiments there are indications that tight, frequently executed loops (like DSP kernels) fit into the buffer completely, which will result in equivalent performance to an uncompressed cache. In addition, some researchers

[23] indicate that similar organization might contribute significantly to low-power design, since the buffer cache filters out power-consuming accesses to the larger L1 cache. The structure of the whole system is depicted in Figure 11. The pipeline stages are outlined on the diagram and detailed cycle count assumptions can be found in the Table 2 of the Appendix. To

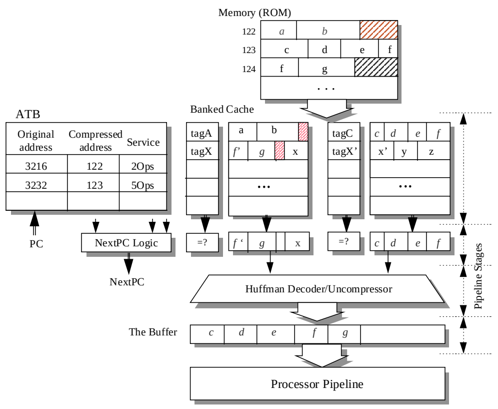

Figure 11. ICache for Compressed Encoding.

summarize, the differences of this cache from the baseline cache are: an additional pipeline stage on the hit path (decompressor), the L0 buffer, and the different (restricted) placement policy.

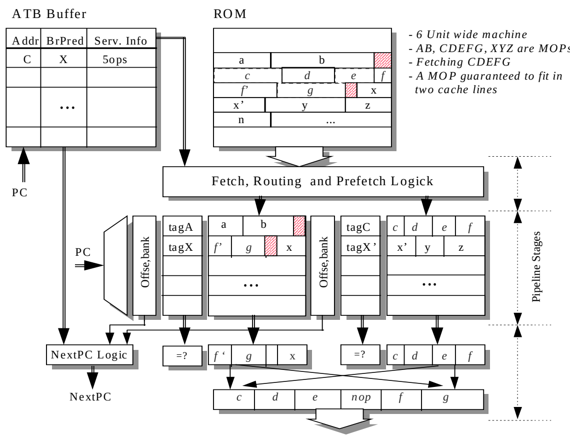

Figure 12. The ICache for Tailored encoding

5 The ICache Design for the Tailored ISA

The objectives for the tailored ISA cache are quite different from the compressed encoding cache: ops are stored in a form ready for consumption by the core decoder. However, it also uses the Banked Cache as its core design element. The key difference is the logic in the miss path which is responsible for extraction and placement of MOPs in the storage. It also plays the role of prefetch engine to guarantee that a whole block is residing in the cache according to the restricted placement policy. If parts of a block are overwritten, it takes care of invalidating the rest of it. The overall organization is shown in Figure 12. The hit path (similarly to the Banked Cache) has only one extra stage for alignment of ops. The main difference from the baseline is an extra stage on the miss path and a different placement policy. Branch prediction is still used (associated with the ATB) to ensure high pipeline utilization. For a detailed summary of all the performance penalty assumptions, please refer to the Appendix.

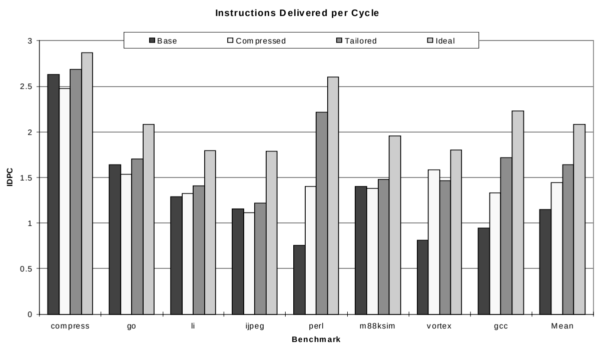

Figure 13. Cache Study Summary

For the experiments we chose moderately sized caches on scale suitable for an embedded system: 16KB, 2-way set associative. The baseline requires a block size that is a multiple of the TEPIC 40bit op size, so its effective size is slightly larger: 20KB, 2-way set associative. All results are summarized in Figure 13. The metric is a measure of instructions (operations) delivered per cycle. The issue width for the core is six operations. The average for the "Ideal" is limited by perfect cache and branch predictor performance. "Base" is for uncompressed code, whereas "Compressed" uses the Full op compression scheme and "Tailored" is for Tailored ISAs. It is particularly interesting to note that both Compressed and Tailored exceed Base on average, although Compressed does worse than Base for several benchmarks (compress, go, ijpeg and m88ksim). This is due to the higher missprediction/miss repair penalties for Compressed compared with Tailored.

The next interesting result is the amount of bus traffic due to instruction cache misses. It is one of the defining factors for power consumption, especially if the ROM is placed on a separate die. In the experiments, power is modeled by counting the number of transactions on the memory bus when bits are *flipped*. The summary is shown in Figure 14. The results track the degree of compression and show savings for Tailored and Compressed over Base. This is because each of the compression schemes brings in more instructions for a given number of bit flips.

One interpretation of the combined results of Figures 10, 13-14, is that Tailored ISA encodings have more advantages than are otherwise obvious from the degree of compression data in Figure 5. Because they do not require an added decoder (as opposed to Huffman-based schemes), there is a net savings in the processor core that can be significant. What is more interesting is that, although Tailored achieves a lower overall cache utilization, the missing extra cycle of branch missprediction penalty more than makes up for this absence in overall performance.

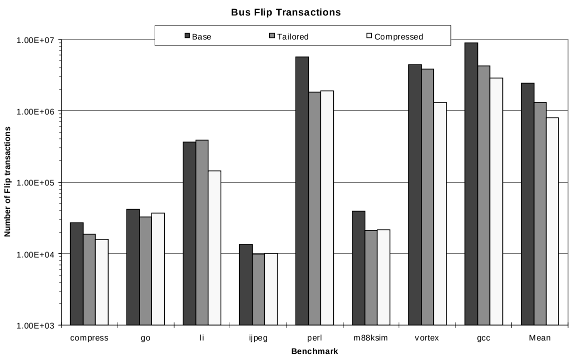

Figure 14. Memory Bus Bit flips Summary.

6 Previous work in depth

In the past there were several similar studies in this area [1,8,9,11,16], as mentioned in the introduction. In several works by Wolfe, et al. [1,16], a technique to execute compressed programs on embedded RISC architectures was studied. Besides diversity in target architecture, the difference from the current study is in the unified approach to compression. While using the Huffman algorithm, one common histogram was built for a set of all experimental benchmarks and only a *byte*-based alphabet was considered. The goal was to create a single encoding for a fixed architecture. This is important when building a general-purpose system but less important for embedded applications. Code is uncompressed on the miss path, but the study does not discuss ICache/IFetch design issues.

Industrial solutions include the IBM CodePack [9], the ARM Thumb [25] and the SGI MIPS16 [26]. The first uses Huffman compression, while the latter two provide special compact subsets of the original ISA. Subset ISAs reduce flexibility, which ultimately results in increased op count and causes slower running applications. CodePack also has the disadvantage of keeping the ICache uncompressed, with the consequences described earlier in this study.

Cooper and McIntosh [11] spend most of their effort reorganizing code at the assembly level via suffix-tree code compression. They reported very moderate compression levels (5-15% reduction). A series of work by Fraser, et al. [8,12] considers elaborate compression algorithms on assembly level code with the same lack of attention to IFetch. The experimental results in this paper show that neglecting IFetch performance may lead to incorrect conclusions about the appropriate scheme to implement.

An interesting study by Liao, et al. [14] employs an effective compression algorithm (External Pointer Model by Storer and Szymanski [22]) on assembly level code with an average of 30% code size reduction. Two implementations, software-only and ‘call-dictionary’ are considered. Both increase the number of branches in the code and (reportedly insignificantly) the op count. Also due to high granularity, some opportunities for compression are missed, and as with CodePack, IFetch uses decompression at miss time.

7 Conclusions

In this paper we have presented a novel approach to reduction of both static and dynamic program size. This approach is to extract the pipeline decoder logic for an embedded processor in software at system development time. The code size reduction is achieved by Huffman compressing or tailor encoding the ISA of the original program. This paper also details the design of IFetch mechanisms for these compression schemes and discusses their performance and cost. Some interesting results were found. In particular, the degree of compression for the ROM doesn’t necessarily translate into an improvement in instructions delivered per cycle. Experiments found that when the missprediction penalty of the added Huffman decoder stage was taken into account, a Tailored ISA approach produced higher performance. Nevertheless, pipeline performance is not always the central goal of embedded systems. Methods like the Full op compression scheme that operates at ICache hit time still achieved median performance advantage over the baseline, while providing significant ROM size savings.

Future work will include consideration of different compression schemes beyond Huffman, the effects of more elaborate branch prediction mechanisms, and usage of complex blocks as fetch units.

References

[1] A. Wolfe, A. Chanin "Executing Compressed Programs on An Embedded RISC Architecture", *In Proc. of 25th International Symposium on Microarchitecture, 1992*

[2] D.A. Huffman "A Method for the Construction of Minimum-Redundancy Codes", *Proc. of the IRE, Vol. 4D, pp. 1098-1101, Sep. 1952*

[3] C. Liem, "Retargetable Compilers for Embedded Core Processors*", Kluwer Academic Publishers, 1997.*

[4] W.A. Havanki "Treegion scheduling for VLIW processor" *MS thesis. Dept. ECE NCSU, Raleigh NC, 1997*

[5] S. Banerjia, W.A. Havanki, T.M. Conte "Treegion scheduling for highly parallel processor" *In Proc. of Euro-Par'97, 1997*

[6] W. A. Havanki, S. Banerjia, T. M. Conte "Treegion Scheduling for Wide Issue Processors*" Proc. of the 1997 4th International Symposium on High-Performance Computer Architecture (HPCA-4), Feb. 1998.*

[7] T.M. Conte, S. Banerjia, S.Y. Larin, K.N. Menezes, S.W. Sathaye, ''Instruction fetch mechanisms for VLIW architectures with compressed encodings"*. In Proc. of the 29th International Symposium on Microarchitecture, pp.201-211, Dec. 1996.*

[8] S. Banerjia, K.N. Menezes, T.M. Conte "NextPC Computation for Banked Instruction Cache for VLIW architecture with a Compressed Encoding" *Technical report Dept. of ECE, NCSU, Raleigh, NC 27695, 1996.*

[8] J. Ernst, W. Evans, C.W. Fraser, S. Lucco T.A. Proebsting "Code Compression" *In Proc. of the '97 International Conf. on Programming Language Design and Implementation, 1997*

[9] M. Game, A. Booker "CodePack: Code Compression for PowerPC Processors", *IBM Microelectronics Division, RTP NC.*

[10] T.M. Conte, "The TINKER Machine Language Manual" *ECE NCSU, Raleigh NC 27695-7911, 1995.*

[11] K.D. Cooper, N. McIntosh "Enhanced Code Compression for Embedded RISC Processors" *In Proc. of the '99 International Conf. on Programming Language Design and Implementation, 1999*

[12] C.W. Fraser "Automatic Inference of models for Statistical Code Compression" In *Proc. of the '99 International Conf. on Programming Language Design and Implementation*, 1999

[13] J.E. Smith "A Study of Branch Prediction Strategies" *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, 1981.

[14] S.Y. Liao, S. Devadas, K. Keutzer "Code density optimization for embedded DSP processors using data compression techniques" In *Proc. of 16th Conference on Advanced Research in VLSI*, (Los Alamitos, CA) 1995.

[15] J.A. Fisher "Trace Scheduling: A Technique for Global Microcode Compaction" *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.

[16] M. Kosuch, A. Wolfe "Compression of Embedded System Programs" *IEEE International Conference on Computer Design*, October 1994.

[17] M. Benes, A. Wolfe, S.M. Nowick "A High-speed Asynchronous Decompression Circuit for Embedded Processors" in *Proc. of the 17th Conference on Advanced Research in VLSI*, 1997.

[18] M.K. Rudberg L Wanhammar "New Approaches to High Speed Huffman Decoding" in Proc. of *ISCAS*, 1996.

[19] D. Alpert, D. Avnon "Architecture of the Pentium Microprocessor" *IEEE Micro, vol. 13, pp. 11-21, June 1993.*

[20] V. Kathail, M. Schlansker, B.R. Rau "HPL PlayDoh architecture specification" *Technical Report HPL-93-80 HP Labs, Palo Alto,CA 1994.*

[21] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter,R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, D.M. Lavery "The Superblock: An effective structure for VLIW and Superscalar compilation" *The Journal of Supercomputing*, vol 7, Jan 1993

[22] J.A. Storer, T.G. Szymanski "Data Compression via Textual Substitution" *Journal of the ACM,* 29(4) pp. 928-951, October 1982.

[23] J. Kin, M. Gupta, W.H. Mangione-Smith "The Filter Cache: An energy efficient memory structure" in *Proc. 30th International Symposium on Microarchitecture*, 1997.

[24] C. Lefurgy, P. Bird, I. Chen, T. Mudge "Improving Code Density Using Compression Techniques" in *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997.

[25] S. Segars, K. Clarke, L. Goudge "Embedded Control Problems, Thumb, and the ARM7TDMI" *IEEE Micro*, October 1995.

[26] K. Kissell "MIPS16: High-density MIPS for the Embedded Market" Silicon Graphics MIPS Group, 1997.

[27] Texas Instruments "TMS320C2x User's Guide", January 1993

Appendix

			Base	***Tailored***	***Compressed***
Next Block prediction *Correct*	Cache Hit	Buffer Hit	1cycle	1cycle	1cycle
		Buffer Miss	1cycle	1cycle	1+(n-1)
	Cache Miss	Buffer Hit	1+(n-1)	2+(n-1)	1cycle
		Buffer Miss	1+(n-1)	2+(n-1)	3+(n-1)
Next Block prediction *Incorrec*t	Cache Hit	Buffer Hit	2cycles	2cycles	1cycle
		Buffer Miss	2cycles	2cycles	2+(n-1)
	Cache Miss	Buffer Hit	8+(n-1)	9+(n-1)	1cycle
		Buffer Miss	8+(n-1)	9+(n-1)	10+(n-1)

Table 1. Cache study cycle count assumptions summary. Note that Base and Tailored do not employ a buffer, which is why Buffer Hit/Miss have no effect

Integer ALU Operation

1	1	2	5	5	5	2	8	5	1	5
T	S	OPT	OPCODE	Src1	Src 2	BHWX	Reserved	Dest	L1	PREDICATE

Integer Compare-to-Predicate Operation

1	1	2	5	5	5	2	3	5	5	1	5
T	S	OPT	OPCODE	Src1	Src2	BHWX	D1	Reserved	Dest	L1	PREDICATE

Integer Load Immediate Operation

1	1	2	5	20	5	1	5
T	S	OPT	OPCODE	Src1	Dest	L1	PREDICATE

Floatin Point Operation

1	1	2	5	5	5	1	6	3	5	1	5
T	S	OPT	OPCODE	Src1	Src2	S/D	Reserved	tssL/U	Dest	L1	PREDICATE

Load Operation

1	1	2	5	5	2	2	1	2	3	5	5	1	5
T	S	OPT	OPCODE	Src1	BHWX	SCS	Res	TCS	Reserved	Lat	Dest	Rsv	PREDICATE

Store Operation

1	1	2	5	5	5	2	2	11	1	5
T	S	OPT	OPCODE	Src1	Src2	BHWX	TCS	Reserved	L1	PREDICATE

Branch Operation

1	1	2	5	5	5	16	5
T	S	OPT	OPCODE	Src1	Counter	Reserved	PREDICATE
0							39

Table 2. Summary of the baseline TEPIC ISA.

Legend:	T	– tail bit to assist Zero-Nop encoding;
	S	– Speculative bit, marking speculated instruction;
	OPT, OPCODE	– Type and Code of the operation;
	Srcx,Destx	– Souce and destination fields;
	BHWX	– Byte/Half-word/Word/Double-word operand types;
	LU	– Lower/Upper part of a register is being accessed;