Merged Dictionary Code Compression for FPGA Implementation of Custom Microcoded PEs

Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski

Center for Embedded Computer Systems

University of California, Irvine

{gorjiara, reshadi, gajski}@cecs.uci.edu

Horizontal Microcoded Architecture (HMA) is a paradigm for designing programmable high-performance processing elements (PEs). However, it suffers from large code size, which can be addressed by compression. In this paper, we study the code size of one of the new HMA-based technologies called No-Instruction-Set Computer (NISC). We show that NISC code size can be several times larger than a typical RISC processor, and we propose several low-overhead dictionary-based code compression techniques to reduce its code size. Our compression algorithm leverages the knowledge of "don't care" values in the control words and can reduce the code size by 3.3 times, on average. Despite such good results, as shown in this paper, these compression techniques lead to poor FPGA implementations because they require many on-chip RAMs. To address this issue, we introduce an FPGA-aware dictionary-based technique that uses the dual-port feature of on-chip RAMs to reduce the number of utilized block RAMs by half. Additionally, we propose cascading two-levels of dictionaries for code size and block RAM reduction of large programs. For an MP3 application, a merged, cascaded, three-dictionary implementation reduces the number of utilized block RAMs by 4.3 times (76%) compared to a NISC without compression. This corresponds to 20% additional savings over the best single-level dictionary-based compression.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-based Systems**]: Microprocessor/ microcomputer applications.
General Terms: Algorithms, Design, Performance, Experimentation.
Additional Keywords: Microcoded Architectures, No-Instruction-Set Computer, Memory optimization, Dictionary-based compression, FPGA.

1. INTRODUCTION

Shrinking time-to-market and high demand for productivity has driven traditional hardware designers to use design methodologies that start from high-level languages. However, meeting design constraints of automatically generated Processing Elements (PEs) is often a challenging and time-consuming task for designers. Moreover, slight changes in the high-level specification require re-running the behavioral synthesis tools, producing a new datapath, and redoing the physical design process. To avoid repeating timing closure and physical synthesis phases, a new generation of custom-PE design technology that is capable of both generating custom datapaths as well as re-programming existing ones (without further modifications) is developed. In these technologies, first a custom datapath is generated for an application, and then the datapath is synthesized and laid out properly to meet timing and physical constraints. The final step is to compile the program on the generated datapath. If the application is changed after synthesis, it is simply recompiled on the existing datapath. This feature significantly improves the productivity of the designer by preventing repetition of physical synthesis phase. Examples of such technology include ARM OptimoDE, No-Instruction-Set Computer (NISC) [1][2], and TIPI [4]. These techniques are targeted for statically-scheduled Horizontal Microcoded Architectures (HMA) [5].

A microcode is a set of bits that controls the units of datapath for one cycle. In statically scheduled HMAs, the compiler compiles the program directly to microcode without using instruction abstraction. HMAs can potentially have better performance, lower power, and lower area than conventional instruction-based processors. This is due to giving the compiler more fine-grained control over the datapath, and hence utilizing datapath resources more efficiently. As a result, highly parallel architectures can be designed as HMA without any concern about the complexity of the controller, hardware scheduler, and instruction decoder. Despite all these benefits, HMAs suffer from "code bloating".

This paper studies FPGA-implementation of soft-core HMA-based PEs. We compare the code size of a new HMA-based design methodology, called NISC, with that of traditional RISC processors. We observed that although NISC PEs outperform typical RISC processors by five times on average, their code sizes are about four times larger than those of RISC. In this paper, we propose several low-overhead dictionary-based code compression techniques to reduce the code size. Our compression algorithm leverages the knowledge of "don't care" values in the control words to improve the compression efficiency and can reduce the code size by 3.3 times, on average. Despite such good results, as shown in this paper, these compression techniques lead to poor FPGA implementations because they require many on-chip RAMs. To overcome this limitation, we propose to merge every two dictionaries into a single dual-port memory unit on FPGAs. Using this approach, the block RAM utilization is improved by 46%. Also, for large applications, we propose using *cascaded dictionaries*, where multi-levels of dictionaries are used to decompress the code. For MP3 application, a merged, cascaded, three-dictionary implementation reduces the number of utilized block RAMs by 4.3 times (76%) compared to a NISC without compression. This corresponds to 20% additional savings compared to the best single-level dictionary-based compression.

This paper is organized as follows: Section 2 presents an overview of NISC Technology. Section 3 presents a motivating example to emphasize the need for code-size reduction techniques in HMAs. Section 4 is an overview of existing code-size reduction techniques. Section 5 discusses our multi-dictionary compression approach, its effectiveness in terms of compression ratio, and its limitation in terms of number of utilized block RAMs. Section 6 introduces our FPGA-aware compression technique that can significantly improve block RAM utilization in FPGAs. Section 7 proposes cascading two-level of dictionaries for code optimization of large program. Section 8 discusses the worst-case performance penalty of decompression. Finally, Section 9 concludes the paper.

2. OVERVIEW OF NISC TECHNOLOGY

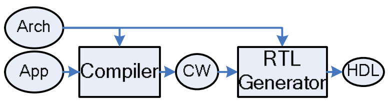

Figure 1- Flow of our toolset

In the NISC design flow [1][2][3], a custom architecture is generated or selected for a given application and then, the program is compiled on the architecture to generate low-level microcodes that we call control words (CW). Finally, the HDL code of the NISC is generated in register-transfer level (RTL) according to the architecture description and the output control words [3]. Our toolset is available online at http://www.cecs.uci.edu/~nisc , where users can specify a new NISC architecture using our Architecture Description Language (ADL) called Generic Netlist Representation

(GNR) [3] and compile their application on it. Also, automatic tools can be used to generate NISC architectures customized for one or more applications [6][7]. NISC customization can be done in three ways: (1) by changing number and type of components and their interconnectivity; [8][7][6] (2) by adding custom functional units (ranging from simple bitwise operations to complex multi-operand operations); [3][9][10] (3) by adding one or more external accelerators [9][10], which may be custom NISCs themselves. The difference between (2) and (3) is that in (2) compiler schedules the communication between the custom unit and the rest of the datapath, while in (3) the software should explicitly define the communication. Customization can significantly improve performance and code size of a design [7], but it is out of scope of this paper. In this paper, we focus only on compression-based code size reduction techniques.

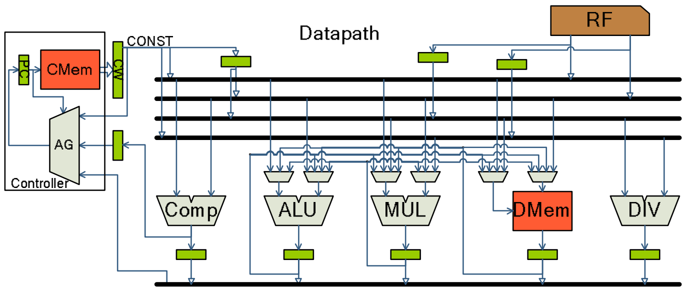

Figure 2- Block diagram of GNISC architecture.

Figure 2 shows an example of a NISC PE, which consists of a datapath and a controller. The datapath contains functional units, register file, registers, multiplexers, and memory. This datapath is general enough to run many applications; therefore, we call it General NISC (GNISC). Our approach relies on a sophisticated compiler [1] to compile a program described in a high-level language to control words that directly drive the control signals of components in the datapath. The CWs are stored in a control memory (CMem) in programmable PEs, or they are synthesized to lookup-table logic in hardwired dedicated PEs. Figure 3 shows the fields of a sample control word. Corresponding to each control signal of each component in the datapath, a field is added to the control words. Also, one or more constant fields are added to store constants in the program. Our compiler generates '0', '1' or "don't care" values for the bits of the control words. A "don't care" value (denoted by 'X') indicates that the corresponding unit is idle at a given cycle and its control signal can be assigned to '0' or '1' without affecting program behavior.

RF_ra0	RF_ra1	RF_wa	RF_we	ALU_op	Mux0_sel	Mux1_sel	R0_load	…	constant

Figure 3- Fields of a control word

Compared to traditional RISC processors, NISC architecture does not have an instruction decode stage. Also, the operations are statically scheduled; hence, it does not have a hardware scheduler. As a result, highly parallel architectures can be designed using NISC without any concern about the complexity of instruction decoder and hardware scheduler.

Similar to NISC, VLIW/EPIC processors [11][12] are also statically scheduled. However, they have several differences:

(1) VLIWs have instruction decoders because they still have the concept of instructions. In fact their decoders are larger than those of RISC processors because they should decode several RISC instructions simultaneously. For example, in TI TMS320C62xx [13] (a modern VLIW architecture), the

instruction decoder consists of two pipeline stages. In contrast, NISC does not need any instruction decode stage.

(2) Adding a new unit to VLIW requires adding a slot to its wide instructions (each slot is equivalent of a RISC instruction). Also the decoder unit must be extended accordingly. However, in NISC, for each new unit, only one or a few control fields are added to the control word. Therefore, more units can be added to the datapath at little cost.

(3) Resource sharing can be implemented more efficiently in NISC than VLIWs, because NISC compiler has low-level control over elements of the datapath. For example, instead of adding two constant fields to the control word of GNISC architecture in Figure 2, one constant field and an *assistant register* is added. If in a given cycle, two constants are needed in the datapath, the compiler can schedule one of the constants a few cycles earlier and transfer it to the assistant register for future use. Similarly, assistant registers are added at the output of register file to reduce the number of required register-file read ports. Such optimizations are to some extend similar to renaming and reservation stations in superscalar processors; but in NISC, they are done at compile-time and do not impose any hardware overhead. In contrast, these optimizations are very challenging in VLIWs because many low-level instructions must be designed for transferring data to the internal registers.

Despite their differences, both NISC and VLIWs have code size issue. The code size reduction techniques proposed for VLIW processors are applicable to NISC as well. Before discussing these techniques, we first present a motivational example that compares the quality and code size of a NISC architecture with those of an instruction-based processor. Since we did not have access to toolset and HDL description of a VLIW processor, we compared NISC to a well-known RISC processor.

3. MOTIVATIONAL EXAMPLE

In this section, we compare the performance and code size of the GNISC with a similar-size RISC processor. Since our toolset generates synthesizable code for Xilinx FPGAs, we choose the Xilinx MicroBlaze for comparing an instruction-based processor with a microcoded one. We synthesized both processors on a Xilinx Virtex4 (90nm) FPGA package using ISE 8.2. We configured MicroBlaze to have an integer multiplier and a divider (no barrel shifter, no floating point unit). Table 1 shows the area (in terms of 4-input LUTs) and clock frequency of the processors. Both processors run at about 100MHz, and occupy nearly the same number of 4-input LUTs.

Table 1- Area and clock frequency of MicroBlaze and GNISC

Processors	Clock freq.(MHz)	# 4-input LUTs
MicroBlaze	105	1581
GNISC	100	1576

We compiled and simulated a set of benchmarks including *dijkstra*, *sha*, *adpcm_coder*, *adpcm_decoder* and *CRC32* from MiBench (the free version of EEMBC embedded benchmarks available at http://www.eecs.umich.edu/mibench), and a fixed-point Mp3 decoder (more than 10,000 lines of C code available at http://www.underbit.com/products/mad). For all benchmark, we have removed file I/O and *printf* calls to make the code suitable for FPGAs. MiBench provides a small and a large input for the benchmarks. We used a subset of small input. We also set the compiler optimizations to the maximum possible level (i.e. –O2) to achieve the best performance with both NISC and MicroBlaze. For each benchmark, to get the accurate execution cycle count, we generated RTL Verilog code of the design and simulated it using Modelsim simulator.

Table 2- Comparing GNISC with MicroBlaze

	MicroBlaze			GNISC			GNISC vs. MicroBlaze	
Benchmarks	#cycles	code size		#cycles	code size		speedup	code size
		KB	#BRAM		KB	#BRAM	(x)	ratio
adpcm_coder	256748693	1.956	1	74321930	6.960	4	3.45	5.10
adpcm_decoder	322766405	1.364	1	63082673	5.075	3	5.12	2.59
CRC32	209436647	1.264	1	21901993	2.567	3	9.56	2.03
dijkstra	25927532	1.928	1	9764682	9.614	6	2.66	4.99
sha	183030479	3.156	2	19282976	14.123	11	9.49	4.47
Mp3	2668445	44.62	21	897452	216.659	117	2.97	4.86
Average							5.54	4.01

Table 2 shows the number of cycles and code size of each benchmark on the two processors. The code size of MicroBlaze (the third column) is the size of instruction section (.text) of the ELF file generated by the compiler. Also the fourth column is the code size in terms of number of on-chip Block RAMs (BRAMs). BRAMs are ASIC memory units that exist on modern FPGA chips. Similarly, the sixth and seventh columns show the code size of GNISC in terms of KB and number of utilized BRAMs. The eighth column shows speedup of GNISC compared to MicroBlaze. The ninth column shows the ratio of GNISC code size (KB) to that of MicroBlaze. On average, GNISC runs 5.54 times faster than MicroBlaze, while its code size is four times larger. Although GNISC occupies almost the same number of LUTs as of MicroBlaze, it needs significantly more BRAMs to store the code. In this paper, we show that different dictionary-based compression techniques can reduce the code size (in terms of KB), but they may fail to reduce the number of utilized BRAMs in FPGA-based implementation. The goal of our code optimization technique is to reduce the code size of NISC processors (both in terms of BRAMs and KB) while maintaining their performance benefits.

4. OVERVIEW OF EXISTING CODE-SIZE REDUCTION TECHNIQUES

In general-purpose processors, the instruction-set abstraction is used to reduce the code size of processors. In RISC processors, designers define 32-bit or 16-bit [14] [15] instructions to encode wide control words. At runtime, the instructions are decoded back to the control words using a hardware decoder. In most processors, one or more pipeline stages are added to the datapath for instruction decoding. As a result, instructions increase branch delay, and hence, affect the performance of the processor (branch delay is the number of cycles between the fetch of a branch instruction and finishing the computation of branch target address). Branch prediction can partially address this issue, but it increases the area of the processor. On the other hand, designing an instruction-set is a very complex and time-consuming task for a typical hardware designer; because the compiler, assembler, linker and instruction decoder must be re-designed to handle custom instructions. To increase the productivity of designers and to give more control to the compiler, in our approach, we eliminate the need for instruction-set design, and compile the application directly to control words. To reduce the code size, instead of using instructions, we directly compress the control words. This way, we replace instruction decode stage(s) with control-words decompression stage(s). In other words, the performance penalty of decompression is the same as the instruction decoder.

As discussed in Section 2, VLIWs have also very large code sizes. In traditional (aka canonical) VLIWs, instructions have several slots, each of which corresponds to an execution unit in the datapath. Due to limited parallelism in the application, at every cycle, some of the units are idle; hence, their corresponding instruction slot is filled with NOP operations. The existence of many NOPs in the code increases its size significantly. To address this issue, modern commercially successful VLIW architectures (such as TMS320C6x [13]) allow more flexible instructions based on the idea of Various Length Execution Set (VLES). In this approach, NOPs are removed as much as possible and a few shortened instructions are packed into one wide fetch packet. Different instruction packing algorithms are discussed in [16] and [17]. For unpacking VLES instructions, a few pipeline stages are added to the processor. For example, in TMS320C6x, fetch/unpacking consists of four pipeline stages; hence, in this processor, fetch and decode need overall six pipeline stages. If compression is applied to VLIWs, it may also need a few more pipeline stages for decompression. Such high number of pipeline stages increases branch delay and area significantly. In contrast, NISC does not have instructions or instruction slot. Therefore, it does not need the instruction decode stages. Also, since it does not have NOP instructions, it does not need packing, and hence unpacking pipeline stages. The only thing it needs is compression, which may add a few pipeline stages to the processor for decompression.

Compression algorithms [18] can be categorized to two main groups: dictionary-based (DBC) and arithmetic (statistic) based (ABC). Also, they can be categorized as fixed length or variable length depending on whether the compressed words (a.k.a. codewords) have the same length or not.

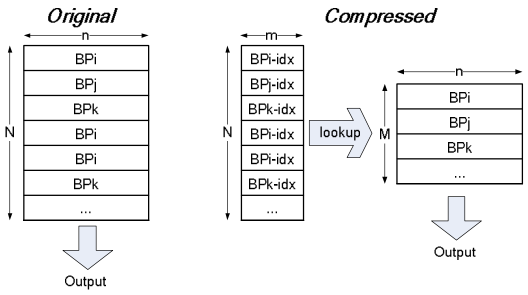

Figure 4-Dictionary-based code compression

The dictionary-based compression techniques rely on the fact that the same patterns appear many times in the data. Figure 4 shows how these techniques typically work. The unique patterns are stored in a dictionary and the original data is replaced with indexes to the dictionary. If original data has N words and each word is n bits, then $N \times n$ bits must be stored. If the number of unique patterns is M, and $m = log_2 M$ is much smaller than n, then, a dictionary based technique can compress the data down to $N \times m + M \times n$ bits, which is often smaller than $N \times n$. However, the decompression costs one extra lookup. To improve the compression, the dictionary-based compression can be combined with Huffman coding where the frequently repeating instructions are placed in low addresses of the dictionary and are coded with fewer number of bits. In this case, the codewords become variable length and need more cycles to decompress. Different variations of dictionary-based compression are proposed for RISC processors. The Code Compressed RISC Processor (CCRP) [19] combines dictionary-based compression with Huffman coding. IBM's CodePack [20][21][22] improves the compression efficiency of CCRP by partitioning instructions to two halves and using two dictionaries to store the unique patterns of each half. In [23], [24], and [25] the concept of dictionary-based compression is extended to sequence of instructions. In these approaches, unique *sequences* of instructions are identified and stored in a dictionary. Dictionary-based compression has also been applied to VLIW processors. In [26], the authors extend the instruction partitioning approach in [20] by automatically partitioning instructions to several fields so that the overall code size is minimized. This approach is applied to traditional (canonical) VLIW processors and shows two to three times reduction in code size. However, in modern (VLES-style) VLIW processors dictionary-based compression is very ineffective (only 10-15% code reduction) because instruction packing increases the number of dictionary entries. To improve its efficiency, in [27], the authors propose combining nearly-identical instructions (that are different only in a few bits) into a single entry in the dictionary. However, they add a new field to codewords to specify the bits that must be toggled during decompression. This approach improves the compression efficiency by a few percentages. Since dictionary-based compression is less effective for modern VLIW processors, arithmetic-based compression techniques (ABC) are proposed [29][30][31], which have better compression ratio but have significantly higher decompression overhead.

Depending on the decompression overhead, compression algorithms can be implemented in two ways for general-purpose processors [31]:

(1) Pre-cache: where the code is decompressed between main memory and cache. In other words, main memory contains the compressed code, and the cache contains the decompressed code. In this approach, the decompression penalty is paid only when a cache miss occurs. Compression techniques that have relatively high decompression overhead (more than one or two cycles) should be implemented as pre-cache. Huffman-coded dictionary based compression techniques [19][20][21][22][23][24][28] and most arithmetic-based techniques [29][30][31] fall into this category.

(2) Post-cache: where the code is decompressed between cache and processor. In other words, both cache and main memory contain compressed code. In this approach, the decompression hardware is on the critical path and should be very fast. Some of the dictionary-based technique [26][27][34] and one of the arithmetic based techniques [31] fall into this category.

In custom reprogrammable hardware units, the entire program and data may fit in on-chip memory blocks. Therefore, cache may impose unnecessary overhead. In our approach, we limit the decompression overhead to one or two cycles in order to allow pre-cache, post-cache, and no-cache implementations. Compared to previous approaches, our approach has several differences:

(1) We leverage the existence of "don't care" values in our binary to improve efficiency of traditional dictionary-based compression techniques.

(2) We show that multi-dictionary techniques (such as [26]) may have good theoretical compression ratio, but when actually implemented on FPGAs, they may occupy more memories than uncompressed code. To address this issue, we propose a technique that leverages on-chip dual-port memory blocks on FPGAs to improve memory utilization.

(3) We also propose a multi-level (cascaded) dictionary architecture that can reduce the number of utilized on-chip memories by an additional 20% compared to single-level dictionary compression.

5. MULTI-DICTIONARY MICROCODE COMPRESSION

In this section, we describe the concept of multi-dictionary compression and explain how "don't care" values in microcode can be leveraged for code size reduction. Our tool constructs a dictionary of unique control words and, in the executable binary, replaces each control word by its corresponding dictionary line addresses. Figure 5 shows a one-dictionary (*opt1*) code compression approach. The memory structure consists of a *Code lookup table* (*CodeLUT*) and a dictionary. The Program Counter (PC) contains the address of *CodeLUT* and is used to read the next *codeword*. The codeword is then used to read the corresponding control word from dictionary.

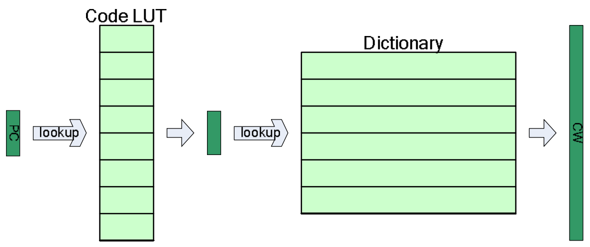

Figure 5- One-dictionary code compression (*opt1*)

The following example shows how dictionary-based compression can reduce the total size of control memory. Suppose that Figure 6 shows the CWs of a sample program. Each CW has 16 bits and the program has nine CWs. Therefore, the code size of the program is 144 bits (16×9). Figure 7 shows the compressed implementation of Figure 6, where the dictionary contains five unique CWs and the CodeLUT contains the corresponding address of the CWs. To address the dictionary, three bits are needed; thus, the codewords are three-bit wide. After compression the total binary size is reduced to 107 (i.e. 3×9+16×5).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0
1	0	0	0	1	1	1	1	0	0	1	0	1	0	1	0
1	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0
1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
0	0	1	0	1	0	1	0	1	0	0	1	0	1	1	0
1	0	0	0	1	1	1	1	0	0	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0

Figure 6- CWs of a sample program

000
001
010
000
011
100
001
011
010

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	0
1	0	0	0	1	1	1	1	0	0	1	0	1	0	1	0
1	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	0	0	1	1	1	1
0	0	1	0	1	0	1	0	1	0	0	1	0	1	1	0

Figure 7- Single-dictionary compression on CWs of Figure 6

00	00
01	00
00	01
00	00
10	10
10	11
01	00
10	10
00	01

0	1	2	3	4	5	6	7
1	0	0	1	0	1	1	0
1	0	0	0	1	1	1	1
0	0	1	0	1	0	1	0

8	9	10	11	12	13	14	15
0	0	1	0	1	0	1	0
1	1	1	0	1	0	1	0
1	0	0	0	1	1	1	1
1	0	0	1	0	1	1	0

Figure 8- Two-dictionary compression on CWs of Figure 6

Since CWs can be very wide with many unique patterns, the dictionary may have many entries and the compression efficiency may be low. To increase the chances of finding matching patterns, we can partition the CWs to smaller slices and construct multiple dictionaries. Usually the total size of the partitioned dictionaries is much smaller than that of a single big dictionary. However, corresponding to each dictionary, a code field must be added to codewords. Figure 8 shows the two-dictionary (*opt2*) implementation of the example CWs of Figure 6. The top dictionary contains the unique patterns of the least-significant half of the CWs, while the bottom dictionary has those of the most-significant halves. Note that the number of unique control word slices in each half is less than the total number of unique control words. The codewords have two fields, which are used to address the two dictionaries. Since each dictionary has four or less entries, the codeword fields are only two bits. Using two-dictionary implementation the code size is reduced to 92 bits (i.e. 4×9+8×3+8×4).

As number of dictionaries increases, the number of codeword fields increases and eventually cancels out the code size reduction achieved by partitioning. Figure 9 shows two-dictionary (*opt2*) and three-dictionary (*opt3*) code compression approaches. The performance penalty in all cases is the same since the dictionaries are accessed in parallel. In addition to the number of dictionaries, the way '*X*' values are resolved in the binary may affect the efficiency of compression.

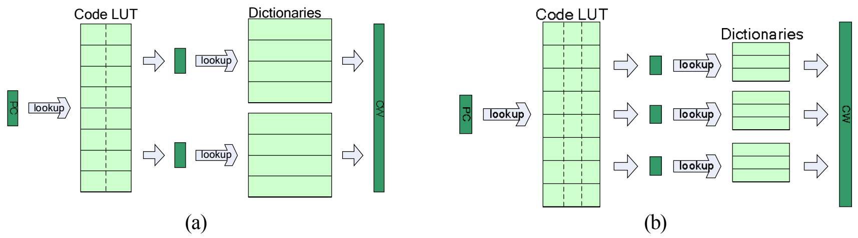

Figure 9- (a) Two-dictionary (*opt2*) and (b) Three-dictionary compression (*opt3*)

5.1 Compression-aware "don't care" resolution (CX)

Figure 10 shows an example of NISC control words. As explained in Section 2, the control words contain "don't care" values (denoted by '*X*'), which indicate that some of the units are idle at a given cycle and their control signals can be assigned to '0' or '1' without affecting program behavior. To build a dictionary for the CWs, one may replace '*X*' values by '0' and then extract the unique patterns. In that case, the dictionary (shown in Figure 11) will have four entries, because only the second and the last vectors match. However, if the '*X*' values are smartly resolved, then seemingly different patterns can be combined into one dictionary entry.

1	0	X	X	1	1	X	X	X	1	1	0	X	0	1
0	X	X	1	1	0	0	0	1	X	0	X	X	1	1
X	0	0	1	X	1	0	1	0	X	X	0	0	0	X
1	X	0	X	1	X	0	X	0	1	1	X	0	X	1
0	0	X	1	1	0	X	0	1	0	0	0	0	1	1

Figure 10- Example of control words generated by NISC compiler

1	0	0	0	1	1	0	0	0	1	1	0	0	0	1
0	0	0	1	1	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	1	0	1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1	1	0	0	0	1

Figure 11- Dictionary content for CWs of Figure 10 ('*X*' are replace by '0')

In general, we need to resolve '*X*' values in CWs so that the total number of unique patterns is minimized. To solve this problem, we convert it to a graph coloring [33] problem. For a given list of bit-vectors, we construct a graph $G(V, E)$, where the vertices in V are the bit-vectors, and the edges in E show the conflict between the vectors. Two bit-vectors v_1 and v_2 do NOT have conflict if they can be collapsed into a single vector:

$$\forall i \in \{1,\ldots, N\},\ v_1[i]=v_2[i] \text{ OR } v_1[i]=\text{'}X\text{'} \text{ OR } v_2[i]=\text{'}X\text{'}$$

where, N is the number of bits in a bit-vector. The edges in E are defined between the vectors that have conflict with each other:

$$E = \{(v_1, v_2) \mid v_1 \text{ has conflict with } v_2\}$$

The algorithm must partition the vertices (or vectors) to sub-categories so that there is no edge (i.e. conflict) between any two vertices in the same category while minimizing the total number of categories. This is exactly the graph coloring problem where each category is represented by a distinct color [33]. Solving the graph coloring problem optimally is NP-hard [35]. But there are many well-known heuristics that generate fine results in polynomial time. We use Welsh and Powell algorithm [33], a greedy heuristic, that (1) sorts vertices based on their degree in decreasing order; (2) traverses the graph and colors as many nodes as possible with color *c*1; and (3) repeats step (2) with color *c*2, *c*3, etc., until no vertex is left uncolored. Figure 12 shows the details of our algorithm. After coloring the graph, a new vector is generated for each color by combining the values of the vectors that share the same color (see line 11-16). Then, all such vectors are replaced with the new vector. The new vectors are also used to fill the dictionary. Figure 13 shows how the algorithm is applied to the example of Figure 10. The graph coloring algorithm produces only two colors. The first, third, and fourth vectors are mapped to the first entry of Figure 14, and the other two vectors are mapped to the second entry.

```
1    Compress (V, N)
2      //Inputs:
3        //V: the list of vectors
4        //N: the bit width of the vectors
5      //output: CV, the list of compact vectors.
6      G = ConstructConflictGraph(V);
7      C = Color(G);        // C is the set of colors
8      for each color c in C
9        // create a new vector that contains only 'X'
10       cv = new BitVector('X', N );
11       for each vector v in V
12         if (color of v is c)
13           //merging the two vectors
14           for i=1 to N
15             if(v[i] != 'X')
16               cv[i] = v[i];

17     for each vector v in CV
18       replace the remaining 'X' in v with '0'
19     CV.Add(cv);
20     return CV;
```

Figure 12- Compression-aware "don't care" resolution algorithm

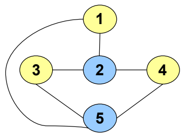

Figure 13- Colored conflict graph of the CWs in Figure 10

1	0	0	1	1	1	0	1	0	1	1	1	0	0	1
0	0	0	1	1	0	0	0	1	0	0	0	0	1	1

Figure 14- Dictionary content using compression-aware '*X*' resolution

This algorithm is coded in C# and is added to our toolset. For our benchmarks, it takes only a few seconds to apply this algorithm. The only exception is MP3, which takes a few minutes.

In [32], we have shown that 'X' values can also be resolved for power optimization at the cost of compromising compression efficiency. We also proposed a profile-driven "don't care" resolution technique that achieves both power and code size efficiency [32].

5.2 Flow of our tool

Figure 15 shows the flow of our tool. First the application is compiled on the datapath and control words are generated. Then, the control words are partitioned and their "don't care" values are resolved using the approach in Section 5.1. Finally, the HDL code of the design is generated. The techniques presented in this paper have been completely implemented and are added to our toolset, which is available online at http://www.cecs.uci.edu/~nisc.

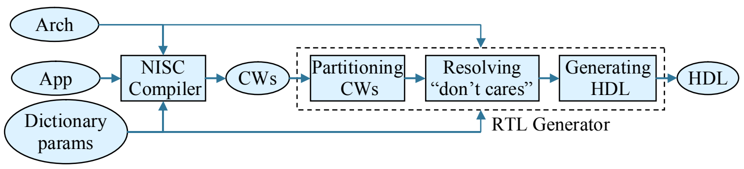

Figure 15- Flow of our tools

5.3 Compression efficiency

In this section we study the effects of multi-dictionary compression as well as compression-aware "don't care" resolution (CX). The same MP3 and MiBench benchmarks used in Section 3 are also used in this Section. Table 3 shows the binary size of the benchmarks compiled on GNISC with and without code compression. The second column (No-opt) shows the baseline code size without any compression. The third to sixth columns (opt1, opt2, opt3, and opt4) show the code size with one to four dictionaries, respectively. In these approaches, the '*X*' values are simply replaced by '0' and then the dictionaries are constructed. As the number of dictionaries increases, the code size (i.e. the total size of dictionaries and *CodeLUT*) of all the benchmarks decreases up to certain points and then increases again. These are the points where the increase in *CodeLUT* size cancels out the benefit of having more dictionaries. The optimum number of dictionaries may vary for different applications.

Table 3- Code size of benchmarks with different compression techniques

		without compression-aware 'x' resolution				with compression-aware 'x' resolution			
	No-opt	**opt1**	**opt2**	**opt3**	**opt4**	**opt1**	**opt2**	**opt3**	**opt4**
adpcm_coder	6.96	4.34	2.90	2.63	2.65	3.77	2.44	2.19	2.19
adpcm_decoder	5.08	3.76	2.38	1.92	2.03	3.24	1.98	1.59	1.68
CRC32	2.57	2.12	1.31	1.08	1.02	1.72	1.04	0.85	0.80
dijkstra	9.61	4.92	3.17	3.02	3.22	4.32	2.71	2.52	2.68
sha	14.12	8.24	5.61	5.23	5.26	6.75	4.45	4.12	4.14
Mp3	216.66	92.66	73.17	79.16	88.98	82.00	63.08	67.66	76.05
average CR	1.00	0.62	0.41	0.37	0.38	0.53	0.34	0.30	0.31

The seventh to tenth columns of Table 3 show the binary size when dictionary-based compression is combined with the compression-aware '*X*' resolution (CX) technique introduced in Section 5.1. The CX technique reduces the code size by an additional 15%-27% (avg. 20%) compared to compression alone. Similarly, as the number of dictionaries increases, the code size of all the benchmarks decreases up to certain points (the highlighted values) and then increases again.

The last row of Table 3 shows the Compression Ratio (CR), a metric commonly used to evaluate a compression algorithm. CR is the ratio between the compressed size and the original size, and smaller CR values show a better compression. On average, for all these benchmarks, the CX-based three-dictionary compression (i.e. *opt3*) outperforms the others and achieves CR of 0.3. In *opt3*, the total code size is one-third of the code size of *No-opt*.

Table 4 compares the code size of compressed GNISC with that of MicroBlaze. The second column shows the code size of optimized GNISC (i.e. the highlighted values in Table 3) and the third column shows the code size of MicroBlaze (from Table 2). The fourth column shows the ratio of the code sizes of GNISC and MicroBlaze. For the small benchmarks, the code size of compressed GNISC is very close or even smaller than that of MicroBlaze. However, for the medium and large benchmarks, GNISC code size is still significantly higher than MicroBlaze (30-40%). In Section 7, we propose a cascaded dictionary structure that can further reduce the code size of larger applications.

Table 4- Comparing code size of compressed GNISC with MicroBlaze

	Code size (Kbytes)		Code size ratio
	GNISC-opt	**MicroBlaze**	**GNISC-opt vs. MicroBlaze**
adpcm_coder	2.19	1.95	1.12
adpcm_decoder	1.59	1.36	1.17
CRC32	0.80	1.26	0.63
dijkstra	2.52	1.93	1.31
sha	4.12	3.16	1.30
Mp3	63.08	44.62	1.41
average			1.16

In this section, we showed that multi-dictionary compression along with compression-aware 'X' resolution reduce the code size by more than three times (i.e. compression ratio of 0.3). However, in the next section we show that low compression ratio does not necessarily result in an efficient design when targeting FPGA platforms. In fact, in some cases, the compressed code may occupy more resources than the uncompressed one.

5.4 Block RAM utilization in FPGA-based implementations

In this section, we investigate whether dictionary-based compression can actually reduce resource utilization in an FPGA-based implementation. In FPGAs, the *CodeLUT* and dictionaries may be implemented using lookup tables or memory blocks (RAM). In today's FPGAs, tens or even hundreds of compact and fast memory blocks exist. Each block has a predefined size and a set of configurations: for example, in Xilinx Virtex4 FPGA, each block RAM is 18Kbits and can be configured statically to a 1×18Kb, 2×9Kb, 4×4Kb, 8×2Kb, 16×1Kb, or 32×0.5Kb memory. These configurations are called RAM *primitives*. In FPGAs, logical memories are implemented using one or more block RAMs depending on their width, depth, and available primitives. Reducing the number of utilized BRAMs is important because it allows packing more processing elements into smaller, low-cost FPGAs.

Table 5 shows the number of utilized 18Kbit block RAMs in different implementations on Xilinx Virtex4SX35. This package is large and contains hundreds of block RAMs. In the MicroBlaze implementation (the second column), most of the benchmarks need only one block RAM for their code, except for *sha* and *Mp*3, which need 2 and 21 blocks, respectively. These numbers are significantly higher for NISC (the third column), because the CWs are wide, and block RAM primitives do not support wide words. In terms of block RAM utilization, NISC requires on average five times more blocks than MicroBlaze. Surprisingly, the compression techniques even increase the number of utilized block RAMs for the smaller applications (i.e. adpcm_coder, adpcm_decoder, and CRC32). However, for medium and large applications (i.e. dijkstra, sha, and Mp3), the compression techniques reduce the number of block RAMs. Although our code compression techniques reduce the code size, they tend to increase the number of memory units, thus occupying more partially-utilized block RAMs. Note that *opt3* is the best compression technique in terms of compression ratio (as shown in Table 3). However, it wastes many block RAMs in FPGA implementations (as shown in Table 5). To address this issue, we introduce our FPGA-aware microcode compression in the following section.

Table 5- Number of utilized 18Kbit block RAMs.

	Memory size (number of 18Kbit block RAMs)					
	MBlaze	**No-opt**	**opt1**	**opt2**	**opt3**	**opt4**
adpcm_coder	1	4	5	6	6	7
adpcm_decoder	1	3	4	5	5	6
CRC32	1	3	4	5	5	6
dijkstra	1	6	5	6	7	9
sha	2	11	5	6	9	11
Mp3	21	117	67	34	38	38

6. FPGA-AWARE MICROCODE COMPRESSION

In Xilinx FPGAs, each block RAM can be configured as single port or dual port with very little logic overhead. We use this property to reduce the number of utilized block RAMs. We integrate every two dictionaries into one dual-port memory. Figure 16(a), (b) and (c) show the dual-port implementation of *opt*2, *opt*3*,* and *opt*4, respectively. Note that, the merged dictionary contents may have more entries than each individual dictionary. However, the size of the merged dictionary is less than the total size of the two dictionaries, because redundant entries can be removed after merging the contents. Since merging dictionaries increases the depth of the dictionary unit, the width of codewords may increase as well. As a result, the total code size most-likely remains the same as before, but the number of utilized block RAMs decreases.

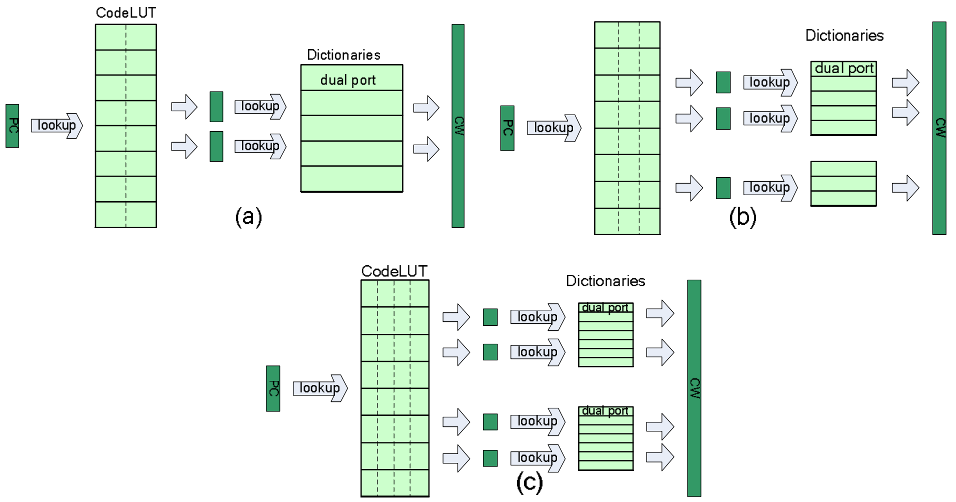

Figure 16- (a) Two-, (b) three-, (c) four-dictionary code compression using a dual-port memory (*opt*2DP, *opt*3DP, *opt4*DP)

Figure 17 shows the dictionary content and *CodeLUT* of the dual-port implementation of Figure 8. The seven entries of the two dictionaries in Figure 8 are compacted to four unique entries in Figure 17. The codewords are also updated to refer to the correct bit patterns. Compared to Figure 8 that requires three block RAMs, Figure 17 requires only two RAMs. Also, the code size is reduced to 68 bits (i.e. 4×9+8×4).

11	00
10	00
11	01
11	00
00	10
00	11
10	00
00	10
11	01

0	0	1	0	1	0	1	0
1	1	1	0	1	0	1	0
1	0	0	0	1	1	1	1
1	0	0	1	0	1	1	0

Figure 17- dual-port memory implementation of Figure 8.

Table 6 shows the number of utilized block RAMs with single and dual port compression techniques. Using dual-port memories (i.e. *opt2DP*, *opt3DP*, and *opt4DP*) reduces the number of utilized block RAMs compared to single-port memories. The minimum number of blocks is achieved using *opt2DP* for *adpcm_coder*, *adpcm_decoder*, *CRC32*, *dijkstra*, and *sha*. For *Mp*3 decoder, however, the minimum is achieved using *opt*2. That is due to a significant size increase in *CodeLUT* of *MP*3 when using dual-port dictionaries. Overall, merging the dictionaries (i.e. *opt2DP*) reduces the number of block RAMs by 46% compared to uncompressed NISC, and by 50% compared to corresponding unmerged dictionary compression (i.e. *opt*2). However, compared to MicroBlaze, *opt2DP* requires 2.3 times more Block RAMs, on average. Note that as the program size increases, the gap between *opt2DP* and MicroBlaze decreases.

Table 6- Number of utilized 18Kbit block RAMs for all compression approaches.

	Memory size (number of 18Kbit block RAMs)								
	MBlaze	**No-opt**	**opt1**	**opt2**	**opt3**	**opt4**	**opt2DP**	**opt3DP**	**opt4DP**
adpcm_coder	1	4	5	6	6	7	3	4	4
adpcm_decoder	1	3	4	5	5	6	2	3	3
CRC32	1	3	4	5	5	6	2	3	3
dijkstra	1	6	5	6	7	9	3	5	5
sha	2	11	5	6	9	11	4	5	5
Mp3	21	117	67	34	38	38	36	42	43

To further improve RAM utilization, we can also adjust the width of dictionaries to match the width of RAM primitives. For example, in a three-dictionary implementation of 75-bit control words, instead of having three equally-sized 25-bit dictionaries, we can have two 32-bit wide dictionaries (that can be merged into one dual-port memory), and an 11-bit dictionary. This way, the block RAMs are more efficiently utilized and deeper memories can be designed with fewer RAMs.

7. CASCADING DICTIONARIES

For large applications, we also propose to use *cascaded dictionaries* where the CodeLUT is replaced by a new dictionary and a narrower CodeLUT. Figure 18 shows the cascaded version of the *opt3DP* (see Figure 16-b). In this approach, the unique three-field codewords are stored in an intermediate dictionary and CodeLUT content is replaced by the references to those dictionary entries. The new codewords must be narrower than the original codewords to achieve savings. This technique requires an additional lookup and hence has higher performance penalty. For small and medium size applications, cascading dictionaries improves the compression ratio but does not reduce the number of utilized Block RAMs. However, for the Mp3 application *cascaded-opt3DP* reduces the number of utilized block RAMs as well (from 34 to 28). If we also adjust the dictionary width to match Xilinx Block RAM primitives, we can reduce the number of blocks to 27, which is 4.3 times better than NISC without compression (No-opt), and 20% better than the best single-level compression technique for MP3 (i.e. *opt*2). Compared to MicroBlaze that needs 21 Block RAMs, cascaded, merged dictionary NISC needs only 28% more RAMs.

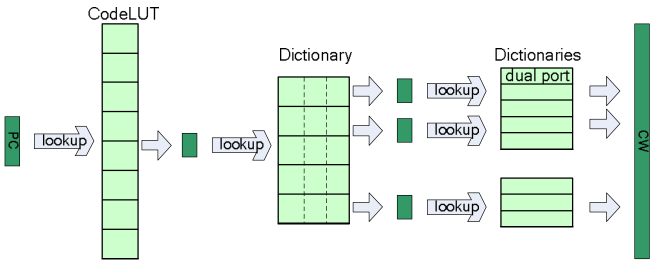

Figure 18- Cascaded version of *opt3DP* in Figure 16(b).

8. PERFORMANCE PENALTY OF DECOMPRESSION

As mentioned in Section 4, the performance penalty of a decompression unit depends on its complexity as well as its position with respect to the cache architecture. If the decompression unit is placed between the main memory and the cache (a pre-cache approach), then decompressions are limited to cache misses only and hence, their overall penalty is negligible. However, if the unit is placed between the cache and the processor (a post-cache approach) or between the memory and the processor (in systems without a cache) then, the penalty is non-negligible. In this section, to determine the worst-case performance penalty, we consider a system without a cache where decompression unit is placed between memory and the processor.

All the compression techniques in Sections 5 and 6 have the same performance penalty. They increase the number of fetch pipeline stages by one, which increases the branch delay by one cycle. The cascaded dictionary structure proposed in Section 7 has a higher performance penalty because it increases fetch pipeline stages (and hence the branch delay) by two. In addition to caches, there are two other approaches for reducing the performance penalty of decompression: (1) branch prediction, and (2) filling branch delay slot. In branch prediction, the controller predicts what would be the target of a branch operation, and starts fetching from the predicted target address. If it turns out that the prediction was wrong, the controller flushes the affected pipeline stages. In the branch delay slot approach, the compiler tries to find operations that are independent of the branch operation, and schedule them after the branch to fill the branch delay slot. The branch delay slot approach is done at compile time and does not impose any hardware overhead. In contrast, branch prediction needs more complex hardware. Currently, our compiler supports branch delay slot, but branch prediction can also be added in the future.

We generated HDL code for the GNISC datapath with and without compression stage(s) and simulated the code using Modelsim simulator. In Table 7, the second and third columns show cycle count of each benchmark without and with compression, respectively. The fourth column shows the performance overhead of single-level compression in terms of the slowdown percentage compared to the baseline GNISC. The performance penalty depends on the number of jump operations and how well the compiler can fill the extra branch delay slot. On average, the performance is degraded by only 9.12% (up to 19%). The fifth column compares the speed of optimized GNISC with that of MicroBlaze processor. The optimized GNISC is on average 5.21 times faster than MicroBlaze. This shows that the compression techniques had little effect on the performance of GNISC even without cache and branch prediction. For cascaded dictionaries (two-level compression), only the MP3 application shows improvement. Therefore, we simulated MP3 and observed an additional 3% performance penalty compared to single-level compression. This penalty is about 8% when compared to uncompressed GNISC.

Table 7- Comparing performance of GNISC-opt with that of GNISC and MicroBlaze

	GNISC (without compression)	GNISC -opt. (with 1-level compression)	GNISC-opt. vs. GNISC	GNISC-opt. vs. MicroBlaze
Benchmarks	#cycles	#cycles	slowdown (%)	speedup (x)
adpcm_coder	74321930	84251684	13.36	3.05
adpcm_decoder	63082673	66504319	5.42	4.85
CRC32	21901993	26008604	18.75	8.05
dijkstra	9764682	10631310	8.88	2.44
sha	19282976	18371827	3.33	9.96
Mp3	897452	927307	4.96	2.88
Average			9.12	5.21

9. CONCLUSION

In this paper, we studied the code size of NISC PEs and compared it with that of traditional RISC processors. We observed that although NISC PEs outperform RISC processors by five times on average, their code sizes are about four times larger.

We studied the use of different variations of dictionary-based code compression techniques on the NISC binary. We showed that although multi-dictionary compression reduces the code size by 3.3 times, its FPGA implementation is very inefficient and can result in occupying more resources than the uncompressed code. To overcome this limitation, we proposed to merge every two dictionaries into a single dual-port memory unit on FPGAs. Using this approach, the block RAM utilization is improved by 46%. Also, for large applications, we proposed using *cascaded dictionaries*, where multi-levels of dictionaries are used to decompress the code. For MP3 application, a merged, cascaded, three-dictionary implementation reduces the number of utilized block RAMs by 4.3 times (76%) compared to a NISC without compression. This corresponds to 20% additional savings over the best single-level dictionary-based compression. Compared to MicroBlaze our compressed MP3 consumes only 28% more block RAMs.

REFERENCES

[1] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", In *Proc. International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS), 2005.

[2] M. Reshadi, B. Gorjiara, D. Gajski, "Utilizing Horizontal and Vertical Parallelism Using a No-Instruction-Set Compiler and Custom Datapaths", In *Proc. International Conference on Computer Design* (ICCD), 2005.

[3] B. Gorjiara, M. Reshadi, P. Chandraiah, D. Gajski, "Generic Netlist Representation for System and PE Level Design Exploration", In *Proc. International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS), 2006.

[4] S. Weber, K. Keutzer, "Using Minimal Minterms to Represent Programmability", In *Proc. International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS), 2005.

[5] A. Agrawala, T. Rauscher, *Foundations of Microprogramming: Architecture, Software, and Applications*, Academic Press, ISBN: 0120451506, 1976.

[6] J. Trajkovic, M. Reshadi, B. Gorjiara, D. Gajski, "A Graph Based Algorithm for Data Path Optimization in Custom Processors", In *Proc. of Euromicro Conference on Digital System Design*, 2006.

[7] B. Gorjiara, "*Synthesis and Optimization of Custom Low-Power NISC Processors*", PhD dissertation, University of California, Irvine, 2007.

[8] B. Gorjiara, D. Gajski, "Custom Processor Design Using NISC: A Case-Study on DCT algorithm", In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia* (ESTIMedia), 2005.

[9] M. Reshadi, D. Gajski, "Interrupt and Low-level Programming Support for Expanding the Application Domain of Statically-scheduled Horizontally-microcoded Architectures in Embedded Systems", In *Proc. Design Automation and Test in Europe* (DATE), 2007.

[10] M. Reshadi, "*No-Instruction-Set-Computer (NISC) Technology Modeling and Compilation*", PhD thesis, University of California, Irvine, 2007.

[11] B. Rau, D. Yen, W. Yen, R. Towle, "The cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs", IEEE Computers, vol. 22, no. 1, pp. 12–34, 1989.

[12] R. CodWell, R. Nix, J. O Donnell, D. Papworth, P. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler", *ACM SIGOPS Operating Systems Review*, vol. 21 , no. 4, 1987.

[13] TMS320C62xx CPU and Instruction Set: Reference Guide, Texas Instruments, Dallas, 1997.

[14] S. Segars, K. Clarke, L. Goudge, "Embedded Control Problems, Thumb, and the ARM7TDMI", *IEEE Micro*, vol. 15, no. 5, pp. 22–30, 1995.

[15] R. Grehan, "16-bit: The Good, the Bad, Your Options", *Embedded Systems Programming*, 1997.

[16] M. Saghir, "Application-Specific Instruction-Set Architectures for Embedded SDP Applications.", PhD thesis, University of Toronto, 1998.

[17] K. Wang, "Code Compaction for VLIW Instructions", MS thesis, University of Toronto, 2001.

[18] K. Rafail, Universal Compression and Retrieval, Kluwer Academic, ISBN:0792326725, 1994.

[19] A. Wolfe, A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture", In *Proc. International Symposium on Microarchitecture*, 1992.

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

[21] T.M. Kemp, R.K. Montoye, D.J. Auerback, J.D. Harper, J.D. Palmer, "A Decompression Core for PowerPC", IBM Corporation, 1998.

[22] C. Lefurgy, E. Piccininni, T. Mudge, "Evaluation of a High Performance Code Compression Method", In *Proc. International Symposium on Microarchitecture*, 1999.

[23] M. Corliss, E. Lewis, A. Roth, "DISE: a Programmable Macro Engine for Customizing Applications." In *Proc. International Symposium on Computer Architecture* (ISCA), 2003.

[24] C. Fraser, "An Instruction for Direct Interpretation of LZ77-Compressed Programs." Technical Report MSR-TR-2002-90, Microsoft Research, Microsoft Corporation, 2002.

[25] J. Lau, S. Schoenmackers, T. Sherwood, B. Calder, "Reducing Code Size with Echo Instructions", In *Proc. Intl. conference on Compilers, Architecture and Synthesis for Embedded Systems* (CASES), 2003.

[26] N. Ishiura, M. Yamaguchi, "Instruction Code Compression for Application Specific VLIW Processors Based on Automatic Field Partitioning", SASIMI, 1997.

[27] M. Ros, P. Sutton, "A hamming distance based VLIW/EPIC code compression technique", In *Proc. Intl. Conference on Compilers, Architectures, and Synthesis for Embedded Systems* (CASES), 2004.

[28] J. Prakash, C. Sandeep, P. Shankar, Y. Srikant, "A simple and fast scheme for code compression for VLIW processors", In *Proc. Data Compression Conference*, 2003.

[29] Y. Xie, W. Wolf, H. Lekatsas, "A code decompression architecture for VLIW processors", In *Proc. International Symposium on Microarchitecture*, 2001.

[30] Y. Xie, W. Wolf, H. Lekatsas, "Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures", In *Proc. IEEE Intl. ASIC Conference*, 2001.
[31] Y. Xie, W. Wolf, H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding", In *Proc. of Intl. Symposium on System Synthesis* (ISSS), 2002.
[32] B. Gorjiara, D. Gajski, "A Novel Profile-Driven Technique for Simultaneous Power and Code-size Optimization of Nanocoded IPs", In *Proc. Intl. Conference on Computer Design* (ICCD), 2007.
[33] T. Jensen, B. Toft, *Graph Coloring Problems*. Wiley-Interscience, New York, ISBN 0-471-02865-7, 1995.
[34] Y. Xie, W. Wolf, H. Lekatsas, "Profile-Driven Selective Code Compression", In *Proc. Design, Automation and Test in Europe* (DATE), 2003.
[35] M. Garey, D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, ISBN:0-7167-1045-5, 1979.
[36] H. Lekatsas, J. Henkel, V. Jakkula, "Design of an One-Cycle Decompression Hardware for Performance Increase in Embedded Systems", In *Proc. Design Automation Conference* (DAC), 2002.