A Flexible Code Compression Scheme using Partitioned Look-Up Tables

Martin Thuresson, Magnus Själander, and Per Stenstrom

Department of Computer Science and Engineering
Chalmers University of Technology
S-412 96 Göteborg, SWEDEN

Abstract. Wide instruction formats make it possible to control microarchitecture resources more precisely by the compiler by either enabling more parallelism (VLIW) or by saving power. Unfortunately, wide instructions impose a high pressure on the memory system due to an increased instruction-fetch bandwidth and a larger code working set/footprint.
This paper presents a code compression scheme that allows the compiler to select what subset of a wide instruction set to use in each program phase at the granularity of basic blocks based on a profiling methodology. The decompression engine comprises a set of tables that convert a narrow instruction into a wide instruction in a dynamic fashion. The paper also presents a method for how to configure and dimension the decompression engine and how to generate a compressed program with embedded instructions that dynamically manage the tables in the decompression engine.
We find that the 77 control bits in the original FlexCore instruction format can be reduced to 32 bits offering a compression of 58% and a modest performance overhead of less than 1% for management of the decompression tables.

1 Introduction

Traditional RISC-like Instruction-Set-Architectures (ISAs) offer a fairly compact coding of instructions that preserves precious instruction-fetch bandwidth and also makes good use of memory resources. However, densely coded instructions tend to increase the efficiency gap between general-purpose processors (GPPs) and tailor-made electronic devices (ASICs) by not being capable of finely controlling microarchitecture resources. In fact, with the advances in compiler technology it is interesting to let wider instructions expose a finer-grain control to the compiler.

Very-Long-Instruction-Word (VLIW) ISAs do exactly this by exploiting parallelism across functional units, whereas architectures with exposed control such as NISC [1] and FlexCore [2] do it in order to expose the entire control to the compiler, thereby having a potential to reduce the efficiency gap between GPPs and ASICs. In fact, recent VLIW ISAs such as IA-64 [3] use 128-bit instruction

bundles containing three instructions each and FlexCore uses as many as 109 bits per instruction.

The downside of wider instructions, however, manifests itself in at least three ways: a higher instruction-fetch bandwidth, a larger instruction working-set, and a larger static code size. This may lead to higher power/energy consumption as well as lower performance, which may in fact outweigh the gains of more efficient use of microarchitecture resources.

Previous approaches to maintain the full expressiveness of wide instruction formats and yet reducing the pressure on the memory system have been to code frequently-used wide instructions more densely. Mips16 [4] and ARM Thumb [5] provide a more dense alternative instruction set and it is possible to switch between the wide and dense instruction formats. In another approach, a dictionary is provided that expands a densely coded instruction into a wide instruction either by coding a single wide instruction with a denser codeword [6, 7] or by coding a sequence of recurring wide instructions with a denser codeword [8–11]. Regardless of the approach, the drawback of all these schemes is that they can only utilize a fraction of the expressiveness of the wide instruction format either because only a subset is compressed or because of the huge dictionaries needed, which can incur significant run-time costs. Our aim is a more scalable approach that can accommodate large programs.

This paper contributes with a novel code-compression scheme that utilizes the *full* expressiveness of the wide instructions by coding the program in a dense fashion. The decompression engine comprises a set of look-up tables (LUTs), each used to compress a partition of the wide instruction word. The compression is done off-line at compile-time by analyzing what subset of the wide instruction set is used in each basic block through a profiling pass. We present an algorithm for compression of the program using dense instructions and for management of the decompression engine at run-time by changing the dictionary entries on-the-fly and yet keeping the run-time costs low. The end result is a decompression methodology that can utilize the full expressiveness of the wide instruction format with low run-time costs. The paper also presents a methodology for how to configure and dimension the decompression engine under various constraints such as keeping the latency of LUTs at a low level.

Based on the FlexCore [2] architecture, we show that the original 77-bit instruction word can be reduced by 58% with less than 1% percent run-time cost in the number of executed instructions for manipulating the LUTs for a set of media benchmarks from the EEMBC suite [12].

As for the rest of the paper, Section 2 describes our baseline architecture model, the FlexCore architecture, followed by a description of the new compression scheme in Section 3. In Section 4, a method for selection of the configuration of the LUTs is presented, and in Section 5 an algorithm for generating the compressed program is shown. The experimental methodology and results are presented in Sections 6 and 7, respectively. Related works are discussed in Section 8 and the paper is concluded in Section 9.

2 FlexCore

FlexCore [2] is an architecture with exposed control, based on the functional units found in a typical five-stage general-purpose pipeline. The data-path consists of a register file, an arithmetic-logic unit (ALU), a multiplier, a load/store unit and a program-control unit connected to each other using a fully connected interconnect and controlled using a wide control word. Figure 1 shows an illustration of the architecture, with the control on top, and the interconnect at the bottom of the figure. One unique property of the FlexCore architecture is that it is possible to include hardware accelerators in the framework and use the interconnect and the general control to flexibly configure a pipeline out of the available datapath elements. Another novel aspect of FlexCore is that its control space is a superset of a traditional five-stage general-purpose processor (GPP), making it possible to fall back on traditionally scheduled instructions found in load/store architectures such as MIPS R2000, if needed.

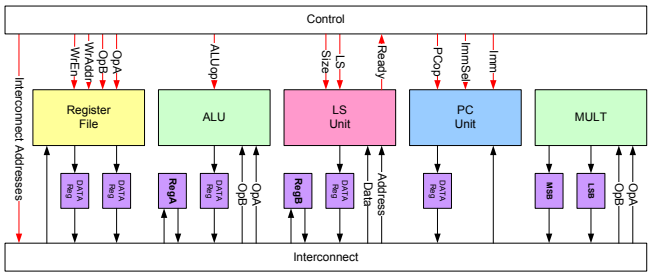


Fig. 1. FlexCore, an architecture using a wide control word, a fully connected interconnect, and the datapath units found in a typical early five-stage load/store architecture such as MIPS 2000.

While this architecture has been shown to be more efficient in terms of execution time and cycle count for embedded benchmarks than a five-stage single-issue pipelined GPP counterpart, the cost in instruction-fetch bandwidth is three times higher compared to a traditional GPP like MIPS R2000, and almost as much in terms of static code size [2]. This makes FlexCore a suitable target architecture for code compression schemes, especially since embedded devices usually have very tight constraints on memory usage and power/energy consumption.

The full control word for the FlexSoC, which consists of 109 signals (out of which 32 comprise the immediate values), can be seen in Table 1. So 77 of the 109 bits are for control, which is the target in this study.

Signal name	Description	Size	Signal name	Description	Size
RegReadA	Reg. A read address	5	PCOp	PC operation	3
RegAStall	Reg. A read stall	1	PCStall	PC stall	1
RegReadB	Reg. B read address	5	I_ALUA	Inter. ALU A	4
RegBStall	Reg. B read stall	1	I_ALUB	Inter. ALU B	4
RegWrite	Reg. write address	5	I_RegWrite	Inter. Reg write	4
RegWE	Reg. write-enable	1	I_LSWrite	Inter. L/S Write	4
Buf1	Buf1 write-enable	1	I_LS	Inter. L/S address	4
Buf2	Buf2 write-enable	1	I_Buf1	Inter. buf 1	4
ALUOp	ALU operation	4	I_Buf2	Inter. buf 2	4
ALUStall	ALU stall	1	I_CtrlFB	Inter. ctrl feedback	4
LSOp	L/S operation	2	MultStall	Mult stall	1
LSSize	L/S size	2	MultEnable	Mult enable	1
LSStall	L/S stall	1	I_MultA	Inter. mult A	4
PC	PC immediate select	1	I_MultB	Inter. mult B	4

Table 1. Control signals in the FlexCore architecture. Size given in number of bits.

3 The Instruction Compression Scheme: Overview

The compression scheme leverages on the fact that during phases of the execution, some combinations of control bits will never appear in the instruction stream. Since the expressiveness found in the wide instructions is thus not utilized, a more efficient encoding scheme can be used. The encoding scheme uses look-up tables (LUTs) to store bit patterns and the compressed instruction is a list of indexes into these tables. The bits found in the tables are then merged to form the decompressed instruction, which can then be executed. Figure 2 shows a decompression structure with four LUTs that together generate the wide instruction. Because of the simple logic involved, and relatively small LUTs needed, we will later show that decompression can be done with virtually no performance overhead as part of the instruction fetch.

The contents of the LUTs can be changed using dedicated *table-manipulating instructions* in the instruction stream. This allows the compiler to use small tables, whose contents are tuned for the particular phase of the execution. The placement of these dedicated instructions will affect the quality of the final solution. The static number of table-manipulating instructions will affect the static code size, whereas the number of table-manipulating instructions in the dynamic instruction stream will affect the performance overhead in terms of instruction overhead and potentially more instruction-cache misses.

In this study, we adopt a straight-forward strategy – a single table-manipulating instruction, called a *LUT-load*, updates a single entry in one look-up table. While this makes our results pessimistic in terms of the overhead caused by the table-manipulating instructions, we note that this is an area for improvement that is subject for future research.

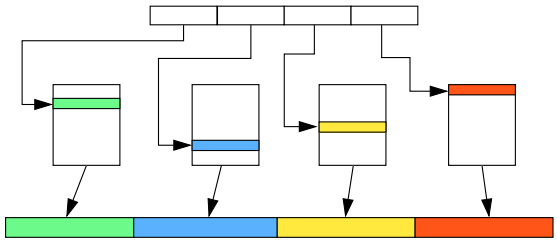


Fig. 2. The instruction-decompression structure encompassing a number of LUTs that expand different parts of the narrow instruction into the wide complete control word.

An important design trade-off in this scheme is the size and configuration of the LUTs. The instruction size depends on both the number of tables, and the size of each. The number of tables dictates the number of indices in the narrow instruction format, and the number of entries dictates the number of bits needed for each index. Also, the size of the tables will influence how often the contents of the tables need to be updated. In the next section, we present a methodology for dimensioning the decompression engine taking this into account.

The methodology also takes into account that many bits in the control word are so called "don't-care" values meaning that they can be set to either zero or one, without affecting the correctness of the program. Don't-care signals have previously been used successfully in the NISC project [6]. The FlexCore-compiler has been updated to generate programs where the "bits" can be 0, 1 or X, giving the compression algorithm additional opportunities for optimizations.

4 A Method for Wide-Instruction Partitioning

In this section, we present a methodology to dimension the compression-engine tables with respect to the number of tables, the number of entries in each table, and the partitioning of the wide instructions across the tables. Since the tables are fixed in the architecture, this design decision needs to be done before fabrication, and is thus done only once.

The method consists of four steps, each one illustrated with examples using the FlexCore control word. In the first step, the designer identifies bits in the control word that are highly correlated and should always be placed in the same LUT. These sets of bits are called *sub-groups*. Table 2 lists the sub-groups identified in the FlexCore architecture.

In the next step, all possible subsets of the set of sub-groups are generated to create possible LUT candidates. A *candidate* is a set of bits that together could become a LUT in the design. An optimization is to already here remove candidates which will not be in the final solution; for example if they are too narrow or too wide. In FlexCore, for example, we might decide to only consider

Sub-group	Signals included	Size	Sub-group	Signals included	Size
RegA	RegReadA + RegAStall	6	Buf	Buf1 + Buf2	2
RegB	RegReadB + RegBStall	6	I_ALUA	I_ALUA	4
RegW	RegWrite + RegWE	6	I_ALUB	I_ALUB	4
PC	PC + PCOp + PCStall	5	I_RegW	I_RegWrite	4
ALU	ALUOp + AluStall	5	I_LS	I_LSWrite + I_LS	8
LS	LSOp + LSSize + LSStall	5	I_Buf1	I_Buf1	4
Mult	MultStall + MultEnable	10	I_Buf2	I_Buf2	4
	I_MultA + I_MultB		I_CtrlFB	I_CtrlFB	4

Table 2. Sub-groups for the FlexCore architecture. Size given in number of bits.

candidates with a width between 7 and 16 bits because of delay and power constraints. Here (RegA, RegB) is a candidate, but (Buf, I_Buf) is too narrow to be a reasonable one, since too many small LUTs lead to a lower degree of compression.

In the third step, LUT candidates are combined into groups called *possible solutions*, so that the candidates in the group cover all the bits in the control word once, and only once. One of many possible solutions in our example is the following candidate list: (I_RegW, I_LS, RegW), (I_ALUB, I_Buf2), (I_ALUA, Mult), (LS, PC), (ALU, I_CtrlFB Buf, I_Buf1) (RegB, RegA).

Finally, each of the possible solutions is evaluated using a user-defined cost function. The cost function evaluates how good the possible solution is for a given application (called workload), and returns a numerical result (lower is better). This makes it possible to find a solution that is relevant for the type of applications that will be executed on the system. In our experiments, we have used cost functions for LUT-access time, compressed instruction width, and energy efficiency. Several cost functions can be given with different priority, and only if a high priority function ties, a lower is evaluated. This makes it easy to add hard design-constraints, such as a maximum access time for the tables, and to make sure that the design fits within a given power envelope.

The cost function will take the LUT configuration into account given by the possible solution, and calculate the LUT sizes needed for the workload, so the whole program can be executed without inserting LUT-loads. In the LUT size calculation, don't-care bits are greedily set to 0 or 1 if it makes it possible to merge two entries into the same value. For example, if the LUT already holds the entry 1X00X, and we try to add the entry X100X, the LUT would be updated to hold the entry 1100X. With this information, the cost can be calculated and returned. Once all the possible solutions have been evaluated, the designer is presented with a list of the solutions with the lowest cost.

While each run of the algorithm gives a list of solutions tuned for that particular workload, we propose running it several times with different workloads. Among all the saved solutions, the designer can pick one that works well for all of the workloads.

One challenge in this methodology is to find a suitable workload. The workload should, for the best solutions, produce LUTs that have a size that results in an acceptable access time and power consumption. For the benchmarks used in this paper, we selected a subset of the full program by running the EEMBC benchmarks one iteration (using the flag *-i1*) and all the program counter-values for the executed instructions were recorded. For each benchmark, the footprint used was the subset of instructions selected by the program-counter trace.

The algorithm also reports the size needed for each suggested LUT. Since the compressed program can change the contents of each LUT, the distribution of the sizes can be used to determine the final sizes for the LUTs.

5 Algorithm for Generation of a Compressed Program

In order to execute a program on a system using our proposed compression scheme, the generated binaries need to be updated by inserting the LUT-load instructions inside the binaries. We have developed an algorithm for generating a compressed program with the LUT-loads placed to keep the performance penalty low. One important approach is to avoid placing LUT-loads in basic blocks that are frequently executed, such as inside inner loops. Since the original wide instructions are compressed, it is possible to reduce the instruction footprint size.

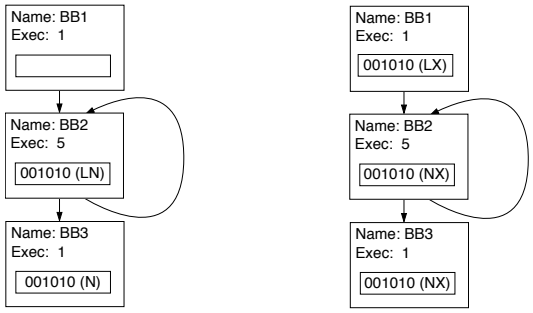
A profiling run is used to get the execution frequency for the basic blocks. We can then estimate the performance overhead caused by the LUT-loads simply by multiplying the number of required table manipulations in each basic block with that block's execution count.

In order to generate the final compressed program, the algorithm uses the uncompressed program, a compiler-generated flow graph showing the relationship between all basic blocks, profiling information at the basic block level, and the LUT-table configuration.

The algorithm uses the flow graph as its main data structure and it associates a complete list of entries with each basic block that are needed in the tables for the basic block to be executed. An entry in this context means a value that is present in a LUT in this basic block. If any basic block requires more entries than the capacity of the tables, the basic block is split into several basic blocks.

The algorithm uses three flags, while updating the flow-graph. The flag L (Load), is used for any entry that does not exist in *all* of the possible predecessors to a given basic block. It is significant since only entries that are marked with L will actually generate LUT-load instructions. The flag N (Needed) shows that an instruction in the basic block requires the entry to be present in the LUT. These entries can never be removed from the basic block, since the code would not work without it. Finally, the flag X (Locked) is used by the algorithm to tag an entry as processed. This makes it easy to process the entries one-by-one, and make sure that all are visited once, and only once.

Initially, all entries in the flow-graph are scanned, and the L and N flags are set according to the description above. Figure 3(a) shows a flow-graph for



(a) Before optimization

(b) After optimization

Fig. 3. Example of a simple flow-graph used to illustrate the algorithm. The graphs show the locations a particular entry in one of the LUTs before and after BB2 is optimized.

a simple loop used to illustrate the algorithm, with each basic block annotated with its execution count from the profiling run. Also, the entry 001010 for one particular LUT is shown with its flags. The entry is initially stored in two different basic blocks, but only marked with L in BB2, since the only path into BB3 is through BB2. The performance penalty for this particular state is five, since a LUT-load in BB2 would be executed five times.

The next step of the algorithm will process the entries marked with L one-by-one, until all have been processed. The main idea behind the algorithm is to look at one entry at a time to see if it is possible, with a lower cost, to make sure the entry already exists in of the possible predecessors. This is done using a recursive optimization function that works as follows.

For each entry that is marked L in the graph, the optimization function tries to "push" the entry to all the predecessors of the basic block. The pseudo-code in Figure 4 shows on a high-level how the function works. It keeps track of the cost as it pushes the entry to the predecessors and returns the best cost it can. The algorithm continues to recursively push the entry further until one of several conditions occur: 1) the value already exists in a basic block; here don't-care values are greedily resolved to zeros or ones, if it helps to find a match 2) the table in the basic block is full; 3) it reaches a root node in the graph; or 4) a maximum recursion depth has been reached. If the minimum cost found is smaller than the current cost, the new entries are added to the affected basic blocks and all the flags are updated. Being able to push entries marked with L out of loops is essential for getting a well performing solution. Since each optimization may increase the number of entries in the LUTs, the entries in the basic blocks that have the most L-flags are processed first.

To keep the execution time of the algorithm down, the recursive depth should be set. This bounds the computational complexity, which would otherwise be

```
// Calculate possible cost achieveble by (recursively) moving
// entry up to all predecessors
optimizeLoad(current_node, entry)
  if(recursionLevel > MAX || current_node.IsRootNode() == 0)
    return INT_MAX
  cost = 0;
  loop over predecessors using pred
    if(pred.HasEntry(entry) cost += 0
    elseif(pred.IsFull())   return INT_MAX // No room to push the entry
    else
      cost_here = pred.ExecutionCount()   // Cost if not pushed further
      cost_push = optimizeLoad(pred, entry)  // Cost if pushed further
      if(cost_here <= cost_push)
         cost += cost_here
      else
         cost += cost_push
         pred.Add(entry) // Add speculatively
  return cost
```

Fig. 4. Simplified pseudo-code for the recursive function. The complete version stores LUT-updates locally and returns the solution.

$O(n^2)$, where n is the number of basic blocks. In our experiments, we have found that very little improvement was found for a depth larger than 30, meaning a load is at most pushed 30 basic blocks.

Figure 3(b) shows the resulting graph after the entry in BB2 has been processed by the recursive function. The execution cost overhead associated with this particular entry has now been reduced to one.

Using the graph it is trivial to generate a compressed program, by issuing LUT-loads at the start of each basic block for each entry marked L. Because of the extra instruction, some branch offsets may have to be recalculated.

6 Experimental Methodology

In order to evaluate our scheme, we have applied the table-configuration and program compression algorithms to the FlexCore architecture and evaluated them using the EEMBC [12] benchmarks Autocorr, FFT, and Viterbi. We have only considered LUT candidates with a table width between 5 and 16 and table sizes that are sufficiently fast and energy efficient according to our cost functions to be defined below.

Included signals	Size (bits)	Included signals	Size (bits)
ALU + I_ALUA + I_ALUB	13	LS + I_LS	13
RegReadA	6	Buf + I_Buf1 + I_Buf2	10
RegReadB	6	PC + I_FB	9
RegWrite + I_RegWrite	10	Mult	10

Table 3. Default LUT configuration used as a baseline for comparisons.

To compare the selected solution with a baseline solution, we use a configuration where signals that, to the best of our knowledge, fit together are placed in the same table. These groups, referred to as the *default* configuration, can be seen in Table 3. In our case study, we combined several cost functions with different priorities to meet timing, size, and power constraints. Since a lower priority function is only evaluated if there is a tie between solutions using a higher priority cost function, it is important to carefully design the cost functions so that ties actually occur. The cost functions used in this study illustrate how this can be done.

The first cost function makes sure that the access time is below one nanosecond for all LUTs. It is designed as a step function, which returns the value one if the delay is below one nanosecond, and a larger value otherwise. The second cost function makes sure we have a sufficiently small instruction format. By dividing the resulting compressed instruction-size by eight (using integer division), the output is quantized. Finally, ties among the solutions with the smallest instructions are broken using a third cost function, which adds the power values of all the tables in the solution. For all functions, a lower cost is better.

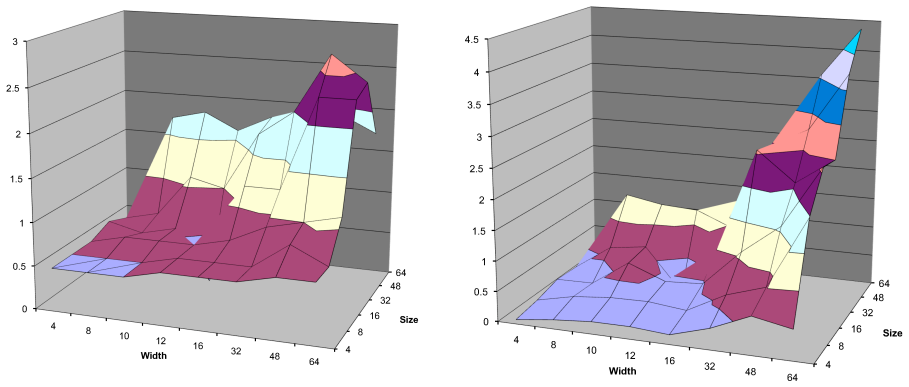
Results on timing and power estimates come from RC-extracted layouts of LUTs of various sizes – depth as well as width have been varied between 4 and 64. The LUTs were described in VHDL and taken through a Cadence Encounter synthesis, placement-and-routing flow for a commercial 65-nm process technology (low-power cell library with standard V_T). The estimates shown in Section 7 were obtained for the worst-case 125 °C corner at 1.1 V.

7 Experimental Results

7.1 Look-up Table Configuration

The data used for the cost functions for delay and power is shown in Figure 5. The graphs show how the LUT dimensions influence power and timing of the decoder. The delay increases with the number of entries in the LUT. This comes as no surprise since the multiplexers for reading and writing data into and out of the LUT need to be wider, thus increasing the logic depth. However, the width also has a negative effect on timing. This is due to longer wires that span across the larger (wider) LUT, which increases the propagation delay. For the power it is the opposite with the width being the more dominant factor. As the LUT becomes wider there are more signals and logic to switch, since there are more bits to read or write in a single cycle, thus increasing the dynamic power. However, an increased number of entries in a LUT also increases the power. This is due to an increase in logic, which results in a higher static power as well as longer wires that require more power for driving them. From Figure 5 one can see that the power dissipation increases rapidly for LUTs wider than 16 bits. The data motivated us to only look at LUTs that are less than 16 bits wide in our case study.

As described in Section 4, the working sets used to decide on a LUT configuration were created by running the program once and only use the executed



(a) LUT Timing (b) LUT Power

Fig. 5. Timing and power estimates from RC-extracted 65-nm layouts of LUTs of various sizes.

instructions. For our benchmarks, only 45% to 60% of the entire set of instructions were executed during such a run. When using this subset of the instructions, our algorithm provided solutions that met our design goals of an access time of less than one nanosecond.

Table 4 shows the delay, number of bits per instruction, and the total power for the default configuration (Table 3) as well as the configuration with the lowest cost found for each of the benchmark subsets. Here we see that the default configuration has a much higher access time than the configuration selected by our algorithm. While the compressed instruction size is similar in the two configurations, the selected configuration is more power efficient.

Benchmark	Delay (ns)	Inst. Width (bits)	Power (mW)
Autocorr (best)	0.6	28	3.8
Autocorr (default)	1.7	27	5.0
FFT (best)	0.6	29	4.8
FFT (default)	1.8	28	4.7
Viterbi (best)	0.6	28	3.8
Viterbi (default)	1.8	27	5.3

Table 4. Results for the table configurations that hold the complete program. Results for both the algorithm-selected configuration (best), and the default configuration are shown.

The next step is to look at the best configuration for each benchmark, and find a one that works overall best, i.e. for all benchmarks. The methodology used was to consider the cost functions for the ten best solutions for each benchmark. Among the possible solutions that had less than one nanosecond latency for all application, we selected the solution that had the narrowest compressed instruction width.

Table 5 shows the selected configuration. For each LUT we also list the width and the total number of entries needed to fit the entire program for each benchmark. With this info, we also list the sizes we use for each table. The sizes were selected to make sure that only 32 bits are needed to index the LUTs. For our configuration, the total number of bits stored in LUTs are only 1036 bits. The final configuration compresses the 77 control signals (109 minus the immediate) down to 32 bits, a compression ratio for each instruction of 58% (41% with the 32-bit immediate included).

Included signals	Group width	Suggested size			Chosen size
		Autocorr	FFT	Viterbi	
I_RegW, I_LS	12	4	5	4	4
I_ALUB, I_Buf2	8	5	6	5	8
I_ALUA, Mult	14	4	8	4	8
LS, PC,	10	15	15	15	16
ALU, I_CtrlFB	9	16	17	16	16
Buf, I_Buf1	6	6	6	6	4
RegW	6	25	28	21	32
RegB	6	18	23	20	16
RegA	6	21	24	20	32

Table 5. Required number of entries for each table for each application and the size chosen for our implementation. The selected sizes are chosen to get a total instruction size of 32 bits for the control bits.

7.2 Program Generation

The next step is to generate code optimized for the selected table-configuration. The benchmarks were compressed using the algorithm outlined above with a maximum recursion level of 30. Table 6 lists the number of "normal" instructions and LUT-loads executed when running the benchmarks. In terms of performance overhead, the results clearly show that the algorithm is successful at placing the LUT-loads. Using the pessimistic calculation that each entry takes one cycle to change, we see that the overhead for changing the contents of the tables is about one percent for all three applications.

Regarding the code size, the static code now contains the compressed instructions *and* the extra table of load instructions. For the three benchmarks,

Benchmark	Dynamic Instruction Count			Static Code Size		
	Normal Instr.	LUT-Loads	Cost	Uncompressed	Compressed	Compr. Ratio
Autocorr	25096	268	1.1%	16.9kB	15.9kB	6%
FFT	169417	2086	1.2%	17.4kB	16.0kB	9%
Viterbi	293755	519	0.6%	15.9kB	14.0kB	12%

Table 6. Compression results for our benchmarks. Cost is the fraction of executed LUT-loads when running our compressed program. Static code size only includes the control bits targeted by our compression.

the static code size targeted by our scheme only decreased by between 6% and 12%, as seen in Table 6. The results can be explained by the fact that our algorithm is optimized for the reduction of performance overhead, and not static code size. In fact, very few LUT-loads were placed in the 90% most frequently executed instructions, leading to a smaller instruction working set for these blocks. An interesting expansion of this work would be to consider other optimization goals for the compression algorithm.

8 Related Work

The NISC project proposes the use of one or two look-up tables to efficiently store programs for FPGA-based custom IPs [6]. While they achieve very high compression ratios (3.3 times for their applications), the size of the tables becomes quite large. Using their approach, with one static LUT, the EEMBC benchmarks evaluated in our study would require between 300 to 400 entries each, making it more suitable in an FPGA environment than to be placed in the latency-critical front-end of an ASIC microarchitecture in which the clock frequency would be heavily constrained.

IBM CodePack [7] is a compression method which enables unmodified cores to run compressed instructions. The instructions are encoded using two tables, one for the op-code and one for the operands. The compressed instructions access the tables using variable length code-words. Because of the complexity of the decompression, a cache is required to hide the decompression latency.

Benini et al. [13] increase the working set that can be placed in the instruction cache by storing the N most frequently executed instructions in a table, and replace them in the code with $log_2(N)$ bit wide codewords. Dictionary-based compression has also been used to replace sequences of instructions with one codeword [10, 11]. While it is not clear how these approaches scale as the instruction width increases and the compiler gets more opportunities to optimize the control word, it is likely that the wider an instruction is, the less likely it is to be reused.

Brorsson and Collin [14] extend previous work in dictionary-based compression by considering dictionary sharing between applications. Similar to our work, they use an instruction for updating the state of the tables, though they only

change the contents of the LUTs during context switches, not between execution phases in the same program.

9 Conclusion and Future Work

This paper presents a novel code compression approach which reduces the pressure on the memory system in wide instruction architectures. Using FlexCore as a case study we show that the scheme, which is based on small look-up tables that each compresses parts of the control word, can compress the 77-bit control-word down to 32 bits with only 1% performance penalty because of the instructions that update the LUTs.

To aid the designer in configuring and dimensioning the look-up tables, a methodology using cost functions is presented. Also, a method to generate compressed programs optimized for a low performance overhead is described.

Accurate profiling information is important for achieving an efficient compressed program. In this study, instruction traces from the execution of the programs were used to get execution counts for each basic block. Another method that could be used to get fast and accurate profiling information is to use sampling techniques developed for feedback-driven compiler optimizations [15].

The compression scheme focuses on compressing the control bits in the instruction word, leaving the immediate field untouched. Frequent value encoding [16] and significance-width compression [17–19] are compression approaches, which have been shown to be very effective at compressing memory traffic [20] and could be a promising approach to use here as well, but this is left for future work.

Acknowledgments

The authors thank Lars Svensson for fruitful discussion on the compression algorithm. We also thank Thomas Schilling for his work on the FlexCore tool-chain and all the members of the FlexSoC project for their contribution to the FlexCore architecture.

References

1. Reshadi, M., Gorjiara, B., Gajski, D.: Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In: Proceedings of the 23rd International Conference on Computer Design (ICCD), IEEE Computer Society (October 2005) 69–76
2. Thuresson, M., Själander, M., Björk, M., Svensson, L., Larsson-Edefors, P., Stenstrom, P.: FlexCore: Utilizing exposed datapath control for efficient computing. To appear in, Journal of Signal Processing Systems (2008) Accepted on the 4th of March 2008.
3. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H., Zahir, R.: Introducing the IA-64 architecture. IEEE Micro **20**(5) (September 2000) 12–23

4. Kissell, K.: MIPS16: High-density MIPS for the Embedded Market. Silicon Graphics MIPS Group (1997)
5. Advanced RISC Machines Ltd.: An Introduction to THUMB. (March 1995)
6. Gorjiara, B., Gajski, D.: FPGA-friendly code compression for horizontal microcoded custom IPs. In: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays (ISFPGA), ACM Press (2007) 108–115
7. Game, M., Booker, A.: CodePack: Code Compression for PowerPC Processors. International Business Machines (IBM) Corporation (1998)
8. Lefurgy, C.R.: Efficient execution of compressed programs. PhD thesis, Ann Arbor, MI, USA (2000) Chair-Trevor Mudge.
9. Lefurgy, C., Piccininni, E., Mudge, T.N.: Reducing code size with run-time decompression. In: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA), IEEE (2000) 218–228
10. Corliss, M.L., Lewis, E.C., Roth, A.: DISE: A programmable macro engine for customizing applications. In: Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA), ACM Press (June 2003) 362–373
11. Thuresson, M., Stenstrom, P.: Evaluation of extended dictionary-based static code compression schemes. In: Proceedings of the 2nd conference on Computing Frontiers (CF), ACM Press (2005) 77–86
12. : EEMBC, the embedded microprocessor benchmark consortium. `http://www.eembc.org` (2008)
13. Benini, L., Macii, A., Nannarelli, A.: Cached-code compression for energy minimization in embedded processors. In: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISLPED), ACM Press (August 2001) 322–327
14. Brorsson, M., Collin, M.: Adaptive and flexible dictionary code compression for embedded applications. In: Proceedings of the international conference on compilers, architectures and synthesis for embedded systems (CASES), ACM Press (2006) 113–124
15. Levin, R., Newman, I., Haber, G.: Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In: Proceedings of the third international conference on High Performance Embedded Architectures and Compilers (HiPEAC). Volume 4917 of Lecture Notes in Computer Science., Springer (January 2008) 291–304
16. Yang, J., Gupta, R., Zhang, C.: Frequent value encoding for low power data buses. ACM Transactions on Design Automation of Electronic Systems **9**(3) (July 2004) 354–384
17. Balakrishnan, S., Sohi, G.S.: Exploiting value locality in physical register files. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE (December 2003) 265–276
18. Brooks, D., Martonosi, M.: Dynamically exploiting narrow width operands to improve processor power and performance. In: Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA), IEEE (January 1999) 13–22
19. Canal, R., González, A., Smith, J.E.: Software-controlled operand-gating. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO), IEEE (March 2004) 125–136
20. Thuresson, M., Spracklen, L., Stenstrom, P.: Memory-link compression schemes: A value locality perspective. IEEE Transactions on Computers **57**(7) (July 2008) 916–927