

# Instruction Set Encoding Optimization for Code Size Reduction

Michael Med      Andreas Krall  
Institut für Computersprachen  
Technische Universität Wien  
Argentinierstr. 8, A-1040 Wien, Austria  
{med,andi}@complang.tuwien.ac.at

**Abstract**—In an embedded system, the cost of storing a program on-chip can be as high as the cost of the microprocessor itself. We examine how much a given application’s program size can be reduced when an instruction set is tailored to the application. We provide different algorithms for calculating an optimized instruction set and evaluate their impact on the size of several benchmark programs. Our results show that an average reduction of 11% is possible, and further improvement can be achieved by changing the instruction length of the given architecture. However compiling other applications with such an optimized instruction set might produce larger code sizes.

## I. INTRODUCTION

In embedded systems the cost of memory used for the instructions can be a significant part of the overall system cost. Therefore, many techniques have been employed to reduce the code size of a program. Examples are the use of procedural abstraction, code compression and tailored instruction sets.

This work focusses on determining the reduction in code size which can be achieved by an optimal encoding of the processor’s instruction set. Our target architecture (xDSPcore [KHPP04], an experimental digital signal processing architecture) supports two instruction sizes – one or two instruction words. Instruction set optimization is achieved by a statical analysis of the compiled program and by giving shorter encodings to frequently occurring instructions. In this paper, we present two algorithms that differ in their run time and in their degree of optimality. Since an instruction set that is optimized for one application will affect the code size for other applications, we also examine this effect.

The next section introduces related work on the subject. In section III, we present the problem and explain the optimization framework. Section IV describes a heuristic approach and section V covers an optimal algorithm based on integer linear programming. The results of a detailed experimental evaluation are presented in section VI.

## II. RELATED WORK

### A. Instruction Set Design

One of the earliest studies into automatic instruction set design was done by Haney [Hay68]. He developed an instruction set design system (ISDS) that is based on a user supplied cost

This work is supported in part by Infineon Technologies Austria and the Christian Doppler Forschungsgesellschaft.

model of the operations and a constraint on the total cost. The ISDS then generates an instruction set by adding features to the operations such that the total value is maximized while staying within the limits. While Haney’s work relied on the user to supply costs and values, Knuth analyzed existing machines to provide data for the design of future machines [Knu71]. He did a comparative analysis of FORTRAN programs from industry and commerce. Knuth examined 440 programs and discovered that the average expression has only two operands, indicating that support for complex instructions is perhaps unjustified.

Sweet and Sandman [SJGS82] analyzed the instruction set of the MESA architecture, a Xerox PARC research project. Their result showed that, on average, the six most frequent instructions made up 50% of the total program size. They reduced the instruction set to 100 generic instructions and added 156 specialized instructions by either combining two operations or combining an operand value with an operation. Evaluation of their work showed an overall reduction in code size of 12%.

Bennet [Ben88] automated this process and applied it to automatically generate a byte-code instruction set for BCPL. In an iterative process, various transformations were applied to the canonical instruction set and the instruction with the greatest predicted code reduction was added to the existing instruction set. Experimental results showed a reduction in code size of about 14%.

Lee et al. [eLCD02] presented an instruction set synthesis technique that employs an efficient instruction encoding method to achieve maximal performance improvement. They built a library of complex instructions with various encoding alternatives and selected the best set of complex instructions while satisfying the instruction bitwidth constraint. They solved the problem by integer linear programming and also by a heuristic algorithm. Evaluation of their algorithm showed improvements of up to 38% over the native instruction set for several benchmark applications.

Holmer [Hol94], [Hol93] introduced new concepts in instruction set design. He presented a tool for assisting in the complete design of instruction sets. His tool took a data path and a set of benchmarks as input and produced as output an instruction set which optimizes a metric. He transformed the set of benchmark programs into a large set of symbolic state transitions each of which represents short code sequences.

Then optimal sets of instructions are determined for each such state transition, and the desired instruction set is the cover of instructions required by the solutions of the benchmark state pairs. Huang and Despain [HD94] built on the work of Holmer and further improved the algorithms in order to generate instruction sets targeted at application specific instruction processors.

Chang et al. [CTM04] proposed a new instruction synthesis paradigm based on a detailed analysis of opcode usage of the MiBench benchmark suite for the ARM Thumb-2 architecture. Their analysis showed that for a wide range of embedded applications it is feasible to utilize a 16-bit instruction format. However each application program may require a different selection of operations and storage components. They suggested that the mapping of instruction set to microarchitecture be delayed until after chip fabrication in order to achieve the highest possible code density while utilizing the fabrication advantages of a mass produced single chip solution.

In [ARK99] Aditya et al. described a mechanism for automatic design and synthesis of very long instruction words (VLIW). The processor design is automatically synthesized into a detailed structural model using VHDL along with an estimate of its area. The system also generates the corresponding detailed machine description and instruction format description that can be used to retarget a compiler and an assembler, respectively. All this is part of an overall design system, called Program-In-Chip-Out (PICO), which has the ability to perform automatic exploration of the architectural design space.

### B. Combined Compiler and Hardware techniques

Liao et al. [LDK99] presented an instruction set architecture (ISA) extension to enhance code compression. The idea is similar to procedural abstraction. They proposed the instruction: CALD *address*, *len*. Their CALD instruction executes *len* instructions at the corresponding address in a hardware dictionary. They built up the hardware dictionary by finding the most common code sequences for a given program. Choosing the code sequences, and an order for the code sequences in the dictionary requires some care, because CALD instructions can execute any substring of the dictionary.

In [LSSC03], Lau et al. further examined Liao's technique which they call *echo* instructions. Two or more similar, but not necessarily identical, sections of code can be reduced to a single copy of the repeating code. The single copy is left in the location of one of the original sections of the code. All the other sections are replaced with a single echo instruction that tells the processor to execute a subset of the instructions from the single copy. They also applied register renaming and instruction scheduling to expose more similarities in code. In order to support these echo instructions efficiently, their work proposes minor architectural modifications to standard processors.

Larin et al. [LC99] presented such a tailored ISA specifically designed to minimize the size of a single program. To generate a tailored ISA, the compiler uses the fewest number

of bits in each instruction encoding that is necessary to satisfy the program's needs. The tailored ISA approach produces compressed binaries that run with low overhead, but the ability to specify custom decoder logic is required.

## III. INSTRUCTION SET ENCODING OPTIMIZATION

### A. Problem formulation

The environment on which we focus is based on an instruction set architecture (ISA) where each instruction is encoded using either a single instruction word (short instruction) or two instruction words (long instruction). The input to our algorithms is an existing instruction set and a set of compiled programs. The goal is to find an encoding of the instructions such that the binary size of the re-compiled input program becomes minimal. Figure 1 shows the concept of our work. A set of programs is compiled using the configurable C-Compiler of the architecture exploration system resulting in an assembler program. From that assembler output, we gather statistics on the usage of the instructions and their operands. Using this data we are able to generate an instruction set optimized to the input program's requirements. Recompiling the input program for this new instruction set results in an assembler program whose binary representation is smaller than the original one.

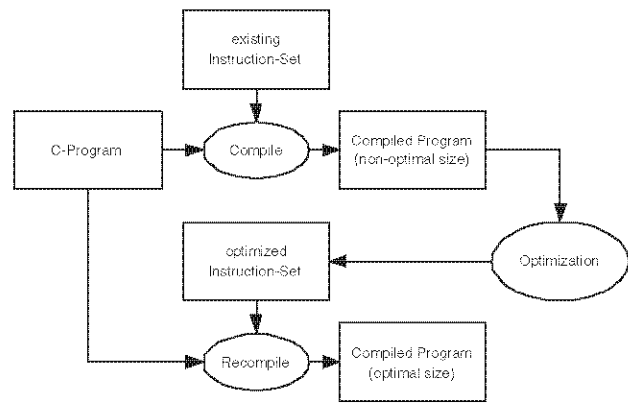


Fig. 1. solution architecture

The major task of the optimization algorithm is to decide whether to encode a specific instruction using a single instruction word or two instruction words. This decision is trivial for instructions which do not fit into a single instruction word. It is also trivial under the assumption that all instructions must not be changed in terms of functionality and the operand sizes. An algorithm similar to Huffman coding is able to find a solution for that reduced problem, i.e. an assignment of instructions to small and large encodings.

We focussed on the broader problem of optimizing the instruction set by creating new instruction variants, i.e. new instructions that are more specific than the original instructions they are based on. The idea is that any such specialized instruction  $\bar{a}$  which implements a subset of instruction  $a$ 's functionality needs fewer bits for encoding. The optimization potential results from the observation that a single program

often uses only a subset of the processor's power. Some functions are never used or, at least, some of them are used more often than others. Immediate values and the sizes of offsets are often smaller than permitted by the original instruction set.

Let's assume that a specific instruction was originally encoded using two instruction words. Optimization might allow us to encode a specialized variant using only a single instruction word. So whenever the new specialized instruction is used instead of the original one we reduce the program's code size by one word.

When the original instruction was already encoded using a single instruction word we can't reduce the code size by introducing a new instruction. But we can free one slot in the single-word encoding space for other instructions by encoding this instruction using two words. This freed encoding space can then be used for a specialized version of another instruction.

### B. Algebraic model

This section introduces an algebraic model of our optimization problem and some notations that will be used later on. Throughout this section we use  $a_i$  for an instruction of the original instruction set  $S$ .  $\bar{a}_{ij}$  is used for instructions of the new instruction set  $\bar{S}$ . The new instruction  $\bar{a}_{i0}$  implements the same functionality as the original instruction  $a_i$ .  $\bar{a}_{ij}$ , ( $j > 0$ ) represents a specialized version of  $a_i$ .

The number of bits required to encode the operands and function flags of any given instruction is denoted as  $b(a_i)$  and  $b(\bar{a}_{ij})$  respectively. We can formulate Lemma 1 which is trivial to prove:

**Lemma 1.** *Let  $a_i$  be an instruction requiring  $b(a_i)$  bits and let  $\bar{S}_i$  be the finite set of all instructions  $\bar{a}_{ij}$  which implement all or part of  $a_i$ 's function; then the following is true:*

$$\forall \bar{a}_{ij} \in \bar{S}_i : b(\bar{a}_{ij}) \leq b(a_i)$$

An instruction set that is encoded using  $n$  bits gives space for encoding  $2^n$  different codes. We will name this *encoding space* or *codespace*. Any instruction requiring  $b$  bits for its operands consumes  $2^b$  bits of this available encoding space. We denote this requirement as  $cs(a_i)$  where

$$cs(a_i) = 2^{b(a_i)}$$

$$cs(\bar{a}_{ij}) = 2^{b(\bar{a}_{ij})}$$

An instruction can be encoded using either a single instruction word of  $w$ -bits or using two instruction words totalling  $W = 2w$  bits. The length of an encoded instruction measured in bits is  $l(a)$ . Whenever we reference the disjoint sets of the original and the new instruction set we will use:

$$S = S_w \cup S_W, S_w \cap S_W = \emptyset$$

$$S_w = \{a | l(a) = w\}, S_W = \{a | l(a) = W\}$$

$$\bar{S} = \bar{S}_w \cup \bar{S}_W, \bar{S}_w \cap \bar{S}_W = \emptyset$$

$$\bar{S}_w = \{\bar{a} | l(\bar{a}) = w\}, \bar{S}_W = \{\bar{a} | l(\bar{a}) = W\}$$

It is obvious that only instructions having  $b(a) < w$ , respectively  $b(\bar{a}) < w$ , can be elements of  $S_w$ , respectively  $\bar{S}_w$ .

The input to our algorithms is a program  $P$  consisting of instructions from the original instruction set. As we are not interested in the order in which the instructions appear we can define  $P$  as a set of tuples. Each tuple contains a reference to the instruction  $a$  and the number of occurrences  $f$ . The same holds for the optimized program  $\bar{P}$ , so that

$$P = \{(a, f) | a \in S, a \text{ occurs } f \text{ times}, f > 0\}$$

$$\bar{P} = \{(\bar{a}, \bar{f}) | \bar{a} \in \bar{S}, \bar{a} \text{ occurs } \bar{f} \text{ times}, \bar{f} > 0\}$$

The size of program  $P$  and  $\bar{P}$  is defined as:

$$|P| = \sum_i l(a_i) f_i, (a_i, f_i) \in P, a_i \in S$$

$$|\bar{P}| = \sum_j l(\bar{a}_j) \bar{f}_j, (\bar{a}_j, \bar{f}_j) \in \bar{P}, \bar{a}_j \in \bar{S}$$

The goal is to find a program-specific transformation  $\Theta_P$  of  $S$  onto  $\bar{S}$  such that the resulting program is smaller in size.  $\Theta_P$  must be valid with respect to the available code space.

$$\Theta : S \rightarrow \bar{S}$$

$$|\bar{P}| = |\Theta(P)| < |P|$$

The optimal transformation of all possible transformations holding the inequality above is denoted as  $\Theta^*$ .

### C. Modifications of the Architecture Exploration System

Part of the architecture exploration system is an optimizing C compiler and an assembler/linker. The assembler/linker is capable of resolving address labels and producing binary code. It also provides simple statistics on the offset lengths used by the instructions.

For our work, the statistics lacked some important details such as data on the usage of an instruction's different variants (e.g. how often a multiply operation is used in an integer or in a floating point context). In order to gather this data we had two choices: either keep the assembler's code untouched and analyze just the binary output – or modify the assembler such that it provides all the data we needed.

We chose a mixed approach and slightly modified the assembler such that it produces *annotated* linked code (see Fig. 2). From this output we were able to gather all the desired statistics without having to write a disassembler specific to the target architecture. To a large extent, this approach is also independent of our architecture exploration system itself. Any system capable of producing similar annotated code can benefit from our tool.

Figure 2 shows an example of such annotated assembler code. The sample consists of two load instructions. The first loads the value from memory addressed by register  $R1$  into register  $D1$ . Afterwards the value of  $R1$  is implicitly incremented. The second instruction is similar – it loads the value in memory addressed by register  $R7$  into the accumulator  $A3$ . Because the instruction set does not contain a simple

```

ld (R1)+, D1      ;original line of assembler
; LOAD
; |00110000000000100001|
; |00piz00f0 [mod] 00aaa [rb] [rb] bbb|
; |f=0 mod=0 i=1 z=0 a=1 rb=0 b=1 p=1|
; |f(+1)=0 mod(+1)=0 i(+1)=1 z(+1)=0 rb(+2)=0|
;*1|LOAD|f(+1)=0 mod(+1)=0 i(+1)=1 z(+1)=0
rb(+2)=0|13|

ldlo (R7 + 0), A3 ;original line of assembler
; LOAD_LONG_OFFSET_ACCU
; |00100010100011100011000000000000000000|
; |0010[mod] 01f1000aaa [rb] bbb00p0ooooooooooooooooo|
; |f=0 mod=0 a=7 rb=0 b=3 p=0 o=0|
; |f(+1)=0 mod(+1)=0 rb(+1)=0 *o(-16)=0|
;*2|LOAD_LONG_OFFSET_ACCU|f(+1)=0 mod(+1)=0 rb(+1)=0
*o(-16)=0|26|

```

Fig. 2. example of annotated assembler output

instruction for this statement, the compiler had to use a more complex instruction that also provides the ability to add an offset to the memory address – in this case the offset is zero.

Lines starting with a semicolon are treated as comments. We use the special sequence `;*`  for annotations important to our statistics gathering. The other lines are for informational purpose to the human analyzing the assembler output.

The `;*`  lines consist of four fields separated by the `|` character. The first field indicates whether this instruction is encoded by a single- or a double instruction word (1 or 2). The second field indicates the instruction’s unique name. It is followed by information about each operand – the operand’s size in parentheses and its actual value. A negative size indicates that this operand holds signed values whereas positive numbers indicate unsigned values. Offset operands have to be treated specially so we mark them with an asterisk for later identification. The last field gives information on the total number of bits this instruction requires. This value includes operands we are not optimizing, e.g. register indices.

When looking at the second instruction of the example in Figure 2, we can learn from the `;*`  line that the `ldlo (R7 + 0), A3` assembler-line is an instance of the `LOAD_LONG_OFFSET_ACCU`-instruction. It is a double-word instruction requiring 26 bits for encoding register addresses, the operands and selecting various functions. The third field lists those bit-fields. The `f`-bit is an unsigned single-bit flag whose value is zero. The same applies to the `mod`- and the `rb`-bits. The `o`-bit-field is marked with an asterisk. It is a 16-bit signed offset field whose value is again zero.

The `LOAD_LONG_OFFSET_ACCU`-instruction instance above is also a good example to examine the potential for optimization. The instruction set designer allocated 16 bits to the offset field. In the example above, its value is zero – which can be encoded using many fewer bits. Is this an extreme sample or are there lots of `LOAD_LONG_OFFSET_ACCU`-instances having a rather small offset? If this is the case, possibly a shorter new variant using only 5 bits might be advantageous. Such a variant would require only 15 bits and would therefore be a candidate for single-word encoding. If

we can replace many of the original double-word instances with this new single-word instruction then we would reduce the program’s size significantly. The algorithms introduced in the following sections try to answer the question whether creating such a new instruction is efficient or not.

#### IV. GREEDY ALGORITHM

The idea behind the greedy algorithm is to select the most promising instructions as single-word candidates for the optimized instruction set. The algorithm has low computational complexity but, in general, does not produce optimal results.

##### A. First Approach

The greedy algorithm implements a straightforward approach to solve the instruction set selection problem. The idea is to encode the most valuable instructions using a single word until all of the available codespace is exhausted. The value of each instruction is calculated on the number of occurrences and the instruction length. An appropriate metric is the quotient of the two, as defined in equation 1 below.  $p(x)$  equals the number of times an instruction occurs (its profit) – and  $w(x)$  is the codespace needed by the instruction (its weight). Instructions that are smaller and occur more often have a larger value and will therefore be selected first.

$$v(x) = \frac{p(x)}{w(x)} \quad w(x) = 2^{b(x)} \quad (1)$$

Our goal is to design a  $n/2n$ -bit instruction set consisting of single word  $n$ -bit instructions and double word  $2n$ -bit instructions. We transform this problem into the equivalent problem of finding an optimal  $2n$ -bit instruction set. Each instruction that is short enough to be encoded using a single instruction word is represented twice. One single-word instance requires  $2^{b(x)+n}$  bits of the  $2^{2n}$  encoding space and a double-word instance requires only  $2^{b(x)}$  bits. We call this total set of instruction candidates the *intermediate representation*. Figure 3 shows an example of a 4/8-bit input instruction set and its intermediate representation.  $X_b$  represents an instruction of length  $b$  bits. Single-word instructions are printed as  $X_b$ . The original instruction set consists of eleven instructions  $A$  to  $N$  – three of which are single words ( $A$ ,  $B$  and  $F$ ). The intermediate representation contains all the original instructions plus a copy for each instruction that might be encoded using a single word (e.g.  $G_7$  is the single-word sibling of the original  $G_3$ ).

The greedy algorithm uses an intermediate representation of Figure 3 as input. At the beginning (step 0), all double-word instructions are selected. The result at this stage is an instruction set containing all original instructions encoded as  $2n$ -instruction words – even if they were single-word encoded in the input instruction set. Then the algorithm selects the most valuable element of all single-word instructions that fit into the remaining codespace.

##### B. Refinement

In the previous example we considered only a reduced version of our original problem. We did not create new

Original n/2n instruction set :

bits	instructions
5	$N_5$
4	$M_4$
3	$G_3, H_3, I_3$
2	$B_2, C_2, D_2, E_2, F_2$
1	$A_1$

Intermediate 2n instruction set :

bits	instructions
7	$G_7, H_7, I_7$
6	$B_6, C_6, D_6, E_6, F_6$
5	$A_5, N_5$
4	$M_4$
3	$G_3, H_3, I_3$
2	$B_2, C_2, D_2, E_2, F_2$
1	$A_1$

Fig. 3. original and intermediate instruction set

variants of instructions that implement a subset of an original instruction's functionality. Our approach for solving this more complex situation is introduced below.

In reality, our algorithm has to consider situations when a candidate instruction is a sub- or a superinstruction of another one that is possibly already selected. Consider the following example: an input program uses the original instruction  $MOV\ Rx, imm\ ed(5)$  20 times. Fifteen out of these 20 times, a reduced instruction with a smaller immediate is sufficient due to the fact that the offset value is always smaller than 8. Therefore we can introduce a new candidate  $MOV\ Rx, imm\ ed(3)$  which needs to be considered when building an optimized instruction set. In the following, we will name such shorter instruction instances  $X_b^*(n)$  and their longer variant  $X_B(N)$  where  $B > b$  and  $N > n$ . Whenever  $X_b^*(n)$  is selected the value of  $X_B(N)$  must be recalculated. This follows from the fact that  $X$ 's profit becomes smaller as soon as one of its subinstructions  $X^*$  has already been selected. Equation 2 shows the modified value function  $v(x)$ . The profit of  $x$  is reduced by the profit of all shorter instructions  $x_j^*$  that are already selected. On the other hand, if  $x$  enters the selection, all previously selected shorter instructions  $x_j^*$  leave the selection instantly. So the actual codespace requirement of selecting  $x$  must be reduced by the sum of all its previously selected subinstructions.

$$v(x) = \frac{p(x) - \sum_j p(x_j^*)}{c(x) - \sum_j c(x_j^*)} \quad (2)$$

$$x_j^* \in sub(x) \wedge x_j^* \text{ is selected}$$

Similarly, we have set the value of all instructions  $X_b^*(n)$  to zero if  $X_B(N)$  has been selected. The simple reason is that selecting a shorter instruction does not make sense when a longer and more general variant of this instruction has already been selected. This is expressed in equation 3.

$$v(x_i^*) = 0, \quad \forall x_i^* \in sub(x) \wedge x_i^* \text{ is selected} \quad (3)$$

We are now able to give the pseudocode-listing (see figure 4) of the greedy algorithm. The value function  $v(x)$  referenced therein is defined as in equation 2.

```

S = intermediate representation including counts
cs = 22n ... n/2n-bit code requested
R = {x | x ∈ S ∧ x is doubleword}
S = S \ R
WHILE cs <> 0 ∧ S ≠ ∅
DO
  SORT S on v(x) FOR ALL x ∈ S
  x = top(S)
  WHILE S ≠ ∅ ∧ size(x) > cs
  DO
    x = next(S)
  DONE
  R = R ∪ {x}
  S = S \ {x}
  S = S \ sub(x)
  cs = cs - space(x)
DONE
RETURN R

FUNCTION v(x)
  space(x) = 2bits(x)
  FOR ALL y ∈ R | y ∈ sub(x)
    space(x) = space(x) - 2bits(y)
    count(x) = count(x) - count(y)
  RETURN count(x)/space(x)

```

Fig. 4. pseudocode of greedy algorithm

### C. Discussion

As we will see in section VI, the greedy algorithm produces quite good, but not always optimal, results. As soon as the available codespace is too small to hold the next element of the sorted input set, the optimality constraint is violated. Dropping an already selected element and choosing other elements instead might improve the overall result significantly.

The biggest advantage of our greedy algorithm is its running time. In contrast to all the other algorithms discussed later the greedy algorithm is amazingly fast. In theory, its solution might be far away from the optimum but due to the nature of our problem this is rather unusual. Practical results (see section VI) show a difference of less than 1% from the optimum.

### D. Optimizations

The pseudocode of Figure 4 recalculated all values and completely resorted the set  $S$  each time an element was selected. However an element's value and its rank change only when one of its subelements has just entered the result set. Therefore we can leave all other elements' values unchanged and recalculate just the affected elements. The optimized pseudocode is listed in figure 5.

```

S = intermediate representation including counts
cs = 22n ... n/2n-bit code requested
R = {x|x ∈ S ∧ x is doubleword}
S = S \ R
SORT S on v(x) FOR ALL x ∈ S
WHILE cs <> 0 ∧ S ≠ ∅
DO
  REPEAT
    x = top(S)
    S = S \ {X}
  UNTIL S = ∅ ∨ size(x) ≤ cs
  R = R ∪ {x}
  S = S \ sub(x)
  cs = cs - space(x)
  FOR ALL y | x ∈ sub(y)
    REARRANGE y in S on new v(y,x)
  DONE
RETURN R

FUNCTION v(y,x)
  space(y) = space(y) - space(x)
  count(y) = count(y) - count(x)
RETURN count(y)/space(y)

```

Fig. 5. pseudocode of greedy algorithm – optimized

## V. ILP-BASED ALGORITHM

The greedy algorithm is fast, but does not give optimal results. We tried an optimal algorithm based on a 0-1 knapsack but it did not produce a result after several days of computation time. This section describes how to formulate the instruction set optimization problem as an integer linear programming (ILP) problem which can be solved using standard available software, such as *lpsolve*.

### A. Basic ILP maximization problem

Our instruction set allocation problem can be formulated as an ILP. As a first approach, we start by defining the goal function and some basic constraints. Our goal is to minimize the total program size which is equivalent to maximizing the number of single word instructions used by the input program. This goal is expressed in equation (4). For each original instruction, we introduce a binary variable  $\underline{x}_j$  that is 1 if the instruction is encoded as a single word and 0 otherwise. The  $p_j$ s are constants counting how often an instruction occurred in the input program. Equation (5) expresses the fact that our codespace is limited and filled by the instructions – either using constant  $w_j$  bits when encoded as a single word or using  $w_j$  bits when encoded as a double word.

$$\text{maximize } \sum_{j=1}^n p_j \underline{x}_j. \quad (4)$$

$$\sum_{j=1}^n w_j \underline{x}_j + w_j(1 - \underline{x}_j) \leq c, \quad (5)$$

$$0 \leq \underline{x}_j \leq 1.$$

The ILP formulated so far is sufficient for finding an optimal re-encoding of the original instruction set without any new

instruction variants. In the next section, we will develop a more sophisticated model that covers that particular case.

### B. Enhanced ILP maximization problem

For each new specialized instruction, we define a new binary variable  $\underline{x}_{j,k}$ ,  $k \geq 1$ . Original instructions are represented by the binary variable  $\underline{x}_{j,0}$ . Whenever an instruction is encoded as a single word the corresponding variable is 1, and 0 otherwise. Constants  $w_{j,k}$ ,  $k \geq 0$  represent the codespace that is required when encoding an instruction as a single word. We already know that it does not make sense to encode a specialized instruction as a double word as it is already covered by the original instruction. So we only need double word codespace for our original instructions represented by  $w_{j,0}$ . Using these constants and the binary variables  $\underline{x}_{j,k}$  we are able to redefine our codespace constraint as in (6).

$$\sum_{j=1}^n w_{j,0}(1 - \underline{x}_{j,0}) + \sum_{j=1}^n \sum_{k=0}^m w_{j,k} \underline{x}_{j,k} \leq c, \quad (6)$$

$$\underline{x}_{j,k} = \begin{cases} 1 & \text{if single-word encoded} \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

For the definition of our goal function, we have to introduce new variables because we can't use  $\underline{x}_{j,k}$  from above. We create a new variable  $\underline{y}_{j,k}$  and a corresponding constant  $p_{j,k}$ .  $\underline{y}_{j,k}$  is 1 if the instruction or some more general instruction is encoded as a single word (8). We use the notation  $a \sqsupseteq b$  if an instruction variant  $a$  is more general than  $b$ . If  $a = 0$  this is true for all  $b$  – as  $a = 0$  represents the original instruction which is always the most general one.

The constant  $p_{j,k}$  counts all instruction instances of the input program that can be encoded by this specific instruction but not by another more specific one. All  $p_{j,k}$  sum to the number of original instruction instances (9). With these new constants and variables we can define our goal function as in (10).

$$\underline{y}_{j,k} = \begin{cases} 1 & \text{if } \exists s \sqsupseteq k : \underline{x}_{j,s} = 1 \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

$$\sum_{k=0}^m p_{j,k} = p_j \quad (9)$$

$$\text{maximize } \sum_{j=1}^n \sum_{k=0}^m p_{j,k} \underline{y}_{j,k}. \quad (10)$$

In order to complete our ILP model, we need to define some additional constraints that express the relations between the instruction variants. We know that it does not make sense to create a single-word instruction when a more general variant is already single-word encoded. This relation and the corresponding equation for our ILP model are expressed in the first line of (11). The second line expresses the relation between the  $\underline{x}_{j,k}$ : whenever a more general instruction is counted as a single-word instruction then certainly all less general can, too. The third line expresses the fact that whenever an instruction is encoded as a single word we can instantly count it as such.

$$\begin{aligned}
 \forall a \sqsupset b : \underline{x}_{j,a} &\rightarrow \neg \underline{x}_{j,b} &\iff \underline{x}_{j,b} + \underline{x}_{j,a} &\leq 1 \\
 \forall a \sqsupset b : \underline{y}_{j,a} &\rightarrow \underline{y}_{j,b} &\iff \underline{y}_{j,b} - \underline{y}_{j,a} &\geq 0 \\
 \forall a : \underline{x}_{j,a} &\rightarrow \underline{y}_{j,a} &\iff \underline{y}_{j,a} - \underline{x}_{j,a} &\geq 0
 \end{aligned} \quad (11)$$

Up to now, we considered all relations except one. When an instruction is counted as single-word encoded, it must either be encoded as such or one of its more general instructions is. This is expressed in (12).

$$\begin{aligned}
 \forall b : \underline{y}_{j,b} \rightarrow \underline{x}_{j,b} \vee (\exists a \sqsupset b : \underline{x}_{j,a} = 1) &\iff \\
 \left( \underline{x}_{j,b} + \sum_{a \sqsupset b} \underline{x}_{j,a} \right) - \underline{y}_{j,b} &\geq 0
 \end{aligned} \quad (12)$$

### C. ILP problem solvers

ILP in general is NP-complete, but there are quite a few good solvers that are publicly available. The most prominent are:

- `lp_solve` is a MIP (mixed integer programming) solver. `lp_solve` is available under GNU LGPL and is capable of solving linear and integer programming problems. It runs as a standalone program or can be incorporated into other software as a library.
- GLPK (GNU Linear Programming Kit) is a set of routines written in ANSI-C which can be linked as a library for solving linear programming problems. It also includes a branch-and-bound algorithm which can be applied to ILP problems.
- OPBDP is an implementation in C++ of an implicit enumeration algorithm for solving (non)linear 0-1 (or pseudo-Boolean) optimization problems with integer coefficients. It contains a couple of different heuristics to solve the problem more efficiently and can be improved by adding heuristics of your own. It is available as a standalone program or can be linked to applications as a library.

For our work we chose `lp_solve` as it supports a very flexible input syntax and it solves our problem instances within a few seconds. Unexpectedly the runtime of OPBDP which is specialized for nonlinear 0-1 optimization problems was much higher.

### D. Discussion

The ILP approach for solving our instruction set allocation problem is very powerful. In contrast to the algorithms presented earlier, the ILP model allows us to add constraints that can't be easily added to the other algorithms. Consider, for example, the requirement that we want the instruction set to contain a single word `ADD Rx, immmed(3)` instruction whenever a `SUB Rx, immmed(3)` is encoded as a single word. With our ILP model, we only need to add a simple equation representing that constraint:  $\underline{x}_{ADD,3} = \underline{x}_{SUB,3}$ .

Our work is based on an existing instruction set that was designed by experienced architects over a long period. The instruction set consists of 217 multi-purpose instructions; 130 of them are 20-bit instructions and 87 of them are 40 bits long. 58% of the 40-bit instructions require fewer than 20 bits for their operands. These instructions were possibly added to the instruction set by the architect at a later time when the 20-bit codespace was already nearly exhausted. Coding these instructions using a short word would have required a reorganization of the existing instruction set encoding.

Statistics showed that fewer than 50% of the available instructions are ever used. This fact alone theoretically opens the potential for much optimization when we are allowed to create a reduced instruction set containing only the instructions that are actually used. However, only a few of the 51 40-bit instructions that require fewer than 20 bits are used by our benchmark programs. So the potential is rather small in practice.

We used a set of DSP benchmarks for our experimental results. The benchmarks were written in standard ANSI C and compiled using the optimizing C compiler of the architecture exploration system. Table I gives an overview of these compiled benchmark programs and some figures. For each benchmark we listed the number of distinct 20- and 40-bit instructions and the resulting total code size. All benchmark programs represent specific fields of DSP-related applications and to some extent they differ in their instruction set requirements. In order to represent a general purpose DSP application, we created a *synthetic* benchmark which is a combination of all other benchmarks. From table I we learn that 20% to 50% of our programs' code size is comprised of double-word instructions. At best, we can therefore theoretically achieve up to a 25% of reduction in code size when we are able to re-encode all 40-bit instructions using a single 20-bit word. In practice, the optimization potential is somewhat smaller – as we will see later on.

### A. Locally optimized instruction sets

This section describes the results when constructing an optimized instruction set for the DSP-benchmark programs introduced in the previous section.

Table II lists the code size of our benchmark programs using the original instruction set and the resulting code size when recompiling it using one of the newly constructed instruction sets. The results apply to a relaxed problem formulation where unused instructions are not included in the new instruction set. Therefore we call these instruction sets *locally optimized*. In section VI-B, we present results for *globally optimized* instruction sets.

Analysing the data from this figure demonstrates that for some programs the theoretical optimum – i.e. encoding all 40-bit instruction used as single-word instructions – is nearly reached. For example 24.1% of the *rijndael* benchmark program's code size was made up of 40-bit instructions. In principle, any optimization technique that does not change

benchmark program	distinct instructions			code size[bits]			code size[%]	
	20-bit	40-bit	total	20-bit	40-bit	total	20-bit	40-bit
adpcm	38	15	53	10080	3840	13920	72.4	27.6
blowfish	40	15	55	12480	4400	16880	73.9	26.1
cmac	53	23	76	40760	39960	80720	50.5	49.5
dct32	39	13	52	13640	9240	22880	59.6	40.4
dct8x8	38	19	57	13260	5480	18740	70.8	29.2
dot	32	11	43	7460	2560	10020	74.5	25.5
gsm	51	23	74	62740	33360	96100	65.3	34.7
g721	50	13	63	25600	8760	34360	74.5	25.5
ghs	43	18	61	45460	13880	59340	76.6	23.4
rijndael	49	19	68	53340	16960	70300	75.9	24.1
serpent	54	21	75	73160	19840	93000	78.7	21.3
viterbi	42	17	59	14140	8480	22620	62.5	37.5
synthetic	61	27	88	185860	98680	284540	65.3	34.7

TABLE I  
BENCHMARK PROGRAMS - CODE SIZE ANALYSIS

benchmark program	original code size		greedy algorithm code size		exact algorithm code size	
	bits	%	bits	%	bits	%
adpcm	13920	100.0	12800	92.0	12780	91.8
blowfish	16880	100.0	15680	92.9	15620	92.5
cmac	80720	100.0	66360	82.2	65640	81.3
dct32	22880	100.0	20000	87.4	20000	87.4
dct8x8	18740	100.0	16920	90.3	16920	90.3
dot	10020	100.0	9280	92.6	9280	92.6
gsm	96100	100.0	85700	89.2	85580	89.1
g721	34360	100.0	31300	91.1	31220	90.9
ghs	59340	100.0	53980	91.0	53880	90.8
rijndael	70300	100.0	63480	90.3	63160	89.8
serpent	93000	100.0	85240	91.7	84800	91.2
viterbi	22620	100.0	19380	85.7	19300	85.3
synthetic	284540	100.0	258520	90.9	253700	89.2

TABLE II  
BENCHMARK ANALYSIS - 20/40 BIT CODE LOCALLY OPTIMIZED

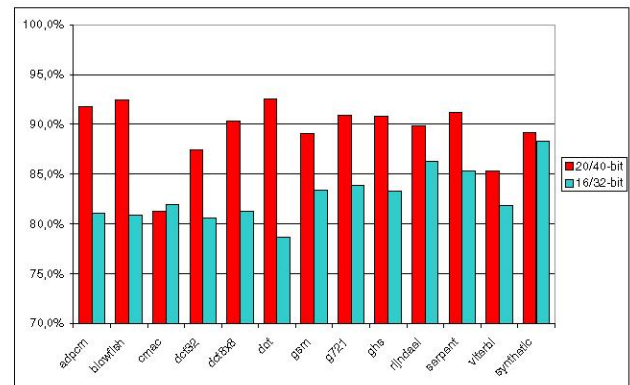


Fig. 6. comparison of 16/32- and 20/40-bit optimization

important architectural features is limited to 12%; with the technique presented here we gained 10.2%.

Figure 6 shows for each benchmark application a comparison of the optimized 20/40-bit and 16/32-bit instruction set. When using single benchmark programs for the optimization, the 16/32-bit encoding results in a significantly smaller code size. When using all benchmark programs together (synthetic) the 20/40-bit and 16/32-bit instruction sets nearly give the same results.

### B. Globally optimized instruction sets

For the data presented in table III, we created for each benchmark program a globally optimized 16/32-bit instruction set using the ILP algorithm. As the resulting instruction set contains all features of the original instruction set, we were able to recompile the other benchmark programs with this instruction set. As can be seen from table III, an instruction set optimized for one program can have a negative impact on the code size of other programs. The only global optimization that produced smaller code for all benchmark programs was the optimization for the synthetic benchmark. All other optimizations produced larger code for at least one other benchmark.

### C. Algorithm execution time

We compared the execution time of the greedy and ILP algorithms. This was interesting as, in general, the ILP-based approach may not finish within reasonable time. The good runtime results of this algorithm are primarily a merit of the lp\_solve application which includes several heuristics and sophisticated algorithms for solving ILP problems efficiently.

Table IV shows the time the two algorithms spent computing a locally optimized 20-/40-bit instruction set. The times were measured on a Pentium III 900 MHz single-processor machine with all algorithms implemented in ANSI-C. As expected, the greedy algorithm is always the fastest one and finishes within a few milliseconds. The ILP based is also fast enough to be used for large problems. If, for very large problems, the ILP algorithm is not fast enough, the greedy algorithm could be used as fall back.

## VII. CONCLUSION

In an embedded system, the cost of storing a program on-chip can be as high as the cost of the microprocessor itself. In this paper we developed a strategy for reducing this cost by reducing the code size of a given application. We achieved this by creating a new instruction set that is optimized



benchmark program	16/32 - instruction set globally optimized for												
	adpcm	blowfish	cmac	dct32	dct8x8	dot	gsm	g721	ghs	rijndael	serpent	viterbi	synthetic
adpcm	84.9	89.3	92.1	92.0	89.7	87.5	89.8	87.2	91.4	89.9	91.4	88.6	88.7
blowfish	94.2	85.8	93.7	100.1	93.3	96.1	94.1	93.6	98.7	93.3	95.7	92.6	94.2
cmac	92.2	90.7	85.1	93.8	91.6	93.9	91.7	91.9	95.3	91.3	91.2	89.1	89.3
dct32	93.6	95.3	94.3	83.8	89.4	92.7	91.0	91.8	94.5	94.3	94.1	92.9	90.2
dct8x8	97.6	96.2	96.9	90.7	84.7	96.5	94.0	93.9	101.3	98.0	98.6	91.7	94.0
dot	84.6	86.2	89.6	87.5	85.0	82.2	88.1	84.8	88.6	87.0	88.9	86.5	87.2
gsm	93.0	94.1	93.1	93.8	93.8	93.7	86.7	90.9	91.4	92.1	91.8	93.9	87.6
g721	94.2	99.3	98.1	102.1	99.5	95.0	95.2	88.7	96.1	97.4	94.5	96.9	92.3
ghs	97.2	99.1	97.6	100.5	102.0	95.7	93.7	94.8	89.8	99.6	95.8	97.9	92.4
rijndael	96.9	99.7	99.0	98.2	99.3	97.7	96.3	94.9	96.2	90.2	93.5	99.2	91.8
serpent	99.8	101.0	99.4	103.9	102.6	100.5	97.9	98.0	99.9	94.8	92.0	99.9	95.5
viterbi	92.6	91.8	91.7	93.9	90.5	93.1	93.7	92.4	96.6	93.5	93.5	85.2	92.7
synthetic	95.9	96.1	94.4	97.4	97.3	96.7	92.3	94.0	94.0	93.6	93.5	95.6	90.5

TABLE III  
BENCHMARK ANALYSIS - 16/32 BIT CODE GLOBALLY OPTIMIZED

	cputime [seconds] of	
	greedy algorithm	ILP algorithm
adpcm	0.07	0.25
blowfish	0.04	0.42
cmac	0.07	0.79
dct32	0.04	0.30
dct8x8	0.05	0.22
dot	0.02	0.20
gsm	0.11	0.83
g721	0.05	0.47
ghs	0.05	0.39
rijndael	0.05	0.42
serpent	0.06	1.27
synthetic	0.17	4.18

TABLE IV  
ALGORITHM RUNTIME FOR LOCALLY OPTIMIZED 16-/32 BIT INSTRUCTION SET

to the application's requirements, exploiting the use of two instruction sizes (one or two instruction words).

Our evaluation showed that, on average, a given program's code size can be reduced by about 11%. Some applications can be reduced in size up to 20%. This was achieved without changing the fundamental architectural components of the underlying processor. Further optimization is possible when the bit-length of the instruction set is reduced. This yielded an improvement of an additional 8% compared to the original code size.

#### ACKNOWLEDGMENT

We like to thank Nigel Horspool and the anonymous reviewers for their helpful suggestions.

#### REFERENCES

[ARK99] Shail Aditya, B. Ramakrishna Rau, and Vinod Kathail. Automatic architectural synthesis of vliw and epic processors. In *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, page 107, Washington, DC, USA, 1999. IEEE Computer Society.

[Ben88] Jeremy Peter Bennett. A methodology for automated design of computer instruction sets. Technical Report UCAM-CL-TR-129, University of Cambridge, Computer Laboratory, March 1988.

[CTM04] Allen Cheng, Gary Tyson, and Trevor Mudge. Fits: framework-based instruction-set tuning synthesis for embedded application specific processors. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 920–923, New York, NY, USA, 2004. ACM Press.

[eLCD02] Jong eun Lee, Kiyong Choi, and Nikil Dutt. Efficient instruction encoding for automatic instruction set design of configurable asips. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 649–654, New York, NY, USA, 2002. ACM Press.

[Hay68] Frederick M. Hayne. "Using a Computer to Design Computer Instruction Sets". PhD thesis, Carnegie-Mellon University, 1968.

[HD94] Ing-Jer Huang and Alvin M. Despain. Synthesis of instruction sets for pipelined microprocessors. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 5–11, New York, NY, USA, 1994. ACM Press.

[Hol93] Bruce K. Holmer. Automatic design of computer instruction sets, 1993.

[Hol94] Bruce K. Holmer. A tool for processor instruction set design. In *EURO-DAC '94: Proceedings of the conference on European design automation*, pages 150–155, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[KHPP04] Andreas Krall, Ulrich Himschrott, Christian Panis, and Ivan Pryanishnikov. xDSPcore: A Compiler-Based Configurable Digital Signal Processor. *IEEE Micro*, 24(4):67–78, July/August 2004.

[Knu71] Donald E. Knuth. An empirical study of fortran programs. In *Software- Practice and Experience*, pages 105–133, 1971.

[LC99] Sergei Y. Larin and Thomas M. Conte. Compiler-driven cached code compression schemes for embedded ilp processors. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 82–92, Washington, DC, USA, 1999. IEEE Computer Society.

[LDK99] Stan Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 4(1):12–38, 1999.

[LSSC03] Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder. Reducing code size with echo instructions. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 84–94, New York, NY, USA, 2003. ACM Press.

[SJGS82] Richard E. Sweet and Jr. James G. Sandman. Empirical analysis of the mesa instruction set. In *ASPLOS-I: Proceedings of the first international symposium on Architectural support for programming languages and operating systems*, pages 158–166, New York, NY, USA, 1982. ACM Press.