Instruction Set Encoding Optimization for Code Size Reduction

Abstract
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 article 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 the impact on the program size of several benchmarks. Results show that an average reduction of 11 percent is possible which can be further improved by changing the instruction length of the given architecture. Compiling other applications with such an optimized instruction set might result in larger code.

Keywords instruction set optimization, code size reduction

1. Introduction
In embedded systems the cost of instruction memory can be a significant part of overall system cost. Therefore, different techniques have been applied to reduce the code size of a program. Examples are the use of procedural abstraction, code compression or tailored instruction sets.

This work focuses on potentials to reduce code size by finding an optimal encoding of the processor’s instruction set. Our target architecture (an experimental digital signal processing architecture) supports two instruction sizes (one or two instruction words). Instruction set optimization is achieved by a statistical analysis of the compiled program and by mapping common instructions to shorter encodings. We will present two algorithms that differ in run time and grade of quality in reaching the optimum. As an instruction set that is optimized for one application affects the code size of other applications we will also examine this impact.

In the next section related work is introduced. In section 3 we present the problem and explain the optimization framework. In section 4 a heuristic and in section 5 an optimal algorithm based on integer linear programming is described. The results of a detailed experimental evaluation are presented in section 6.

2. Related work
2.1 Instruction Set Design
One of the first 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 model of the operations and a constraint on the total cost. The ISDS then generates an instruction set by adding features to the operations trying to maximize the total value 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 [SJS82] analyzed the instruction set of the MESA architecture, a Xerox PARC research project. Their motivation was to reduce the static size of compiled code, since the existing architecture had run out of address space. The basis of their work was a manually designed instruction set consisting of 240 distinct instructions and about 2.5 million instruction bytes of existing code. They analyzed static frequencies of instructions, popular instruction pairs and the distributions of offsets and immediate values. Their result shows that in average the six most frequent instructions make up 50 percent of the total program size. They reduced the instruction set to 100 generic instructions and added 156 specialized instructions 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 percent.

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

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 percent 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 the complete design of instruction sets. His tool takes as input a data path and a set of benchmarks and produces 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 deter-
minded 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, but that each application may require a different selection of operations and storage components. They suggested to delay the mapping of instruction set to microarchitecture to a point 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 word (VL IW), and its generalization, explicitly parallel instruction computing (EPIC). Their project carried out at HP laboratories Palo Alto had similar goals as the project we are working. The major difference is that the EPIC architecture allows arbitrary instruction lengths while our architecture uses only two distinct instruction lengths. 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.

2.2 Combined Compiler and Hardware techniques

The effectiveness of compiler techniques like procedural abstraction to reduce code size is bound by the limitations of the target instruction set. Naturally, these limitations can be removed by modifying the instruction set. The compiler optimizations discussed in this subsection depend on special instructions introduced for the purpose of reducing code size.

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.

Another way to reduce code size is to support variable instruction widths, and allow programs to switch between the different widths during execution. Our instruction set encoding enables the processor to distinguish automatically whether the fetched instruction is e.g. 16- or 32-bits long. Other instruction sets like the MIPS16 or the ARM Thumb have special instructions switching the processor from 16- to 32-bit interpretation mode.

Another approach is to generate a tailored ISA for the program that will execute on the embedded processor. 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 possible to satisfy the program’s needs. For example, if the program uses not more than sixteen integer operations, then four bits will be used for the integer opcode. Similarly, if all the instructions in the program write to one of seven registers, then three bits will be used to encode the destination register. The compiler for their tailored ISA system produces two outputs: a binary written in the tailored ISA, and a HDL description of the decoder for the embedded processor. The tailored ISA approach produces compressed binaries that run with low overhead, but the ability to specify custom decoder logic is required, and additional area is also required by the custom decoder logic.

3. Instruction Set Encoding Optimization

In this section we will describe the basic optimization problem. In the next two sections we will present two different algorithms to solve this problem.

3.1 Problem formulation

The environment we focus on 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.

![Figure 1. solution architecture](image-url)
terms of functionality and their operator’s size. 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 focused on the broader problem of optimizing the instruction set via creating new instruction variants, i.e. instructions that are more specific than the original instructions they are based on. The idea is that any such specialized instruction $a$ which implements a subset of instruction $a$’s functionality needs less bits for encoding. The optimization potential results from the fact 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 sizes of offsets are often smaller than than in 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 when-ever 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 new instructions. But we can free single-word encoding-space for other instructions by encoding this instruction using two words. This freed encoding-space can then be used for specialized versions of other instructions.

3.2 Algebraic model

This section introduces an algebraic model of our optimization problem and some notations that will be used later on. Throughout this chapter we will use $a_i$ for an instruction of the original instruction set $S$. $\pi_{i,j}$ will be used for instructions of the new instruction set $\bar{S}$. The new instruction $\bar{a}_{i,j}$ implements the same functionality as the original instruction $a_i$, $\pi_{i,j}$, $j > 0$ 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 $b(a_i)$ resp. $b(\pi_{i,j})$. We can formulate Lemma 1 which is trivial to prove:

**Lemma 1.** Let $a_i$ be an instruction requiring $b(a_i)$ bits and $\bar{S}$, the finite set of all instructions $\pi_{i,j}$ which implement all or part of $a_i$’s function then the following is true:

$$\forall \pi_{i,j} \in \bar{S}: b(\pi_{i,j}) \leq b(a_i)$$

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 requires $2^b$ bits of this available encoding space. During this work we will denote this requirement as $cs(a_i)$. $cs(a_i) = 2^b(a_i)$

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

$$S = S_w \cup S_{2w}, S_w \cap S_{2w} = 0$$

$$S_w = \{a|l(a) = w\}, S_{2w} = \{a|l(a) = W\}$$

$$\bar{S} = \bar{S}_w \cup \bar{S}_{2w}, \bar{S}_w \cap \bar{S}_{2w} = 0$$

$$\bar{S}_w = \{\pi|l(\pi) = w\}, \bar{S}_{2w} = \{\pi|l(\pi) = W\}$$

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

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

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

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

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

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

$$|\bar{P}| = \sum_j l(\pi_j)f_j, (\pi_j, f_j) \in \bar{P}, \pi_j \in \bar{S}$$

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 $\Theta^*$.

3.3 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 important elements like data on the usage of an instruction’s different variants (e.g. how often a multiply operation is used in an integer or in an 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 every data we needed.

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

![Figure 2. example of annotated assembler output](image-url)
the value in 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 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 parenthesis 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 on, e.g. register indices.

When looking at the second instruction of the example in Figure 2 we can learn from the ;* line that the \texttt{ldlo (R7 + 0), A3} assembler-line is an instance of the \texttt{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 \texttt{LOAD_LONG_OFFSET,ACCU} instruction instance above is also a good example to examine the optimization potential. The instruction set designer allocated 16 bits to the offset-field. In the example above its value is zero - which can be encoded using much less bits. Is this an extreme example or are there lots of \texttt{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 requires only 15 bits and is therefore a candidate for single-word encoding. If we can replace many of the original double-word instances with this new single-word instruction than we are able to 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.

4. 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. In section 5 we will present an optimal algorithm based on integer linear programming.

4.1 First Approach

The greedy algorithm implements a straightforward approach in solving 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 its occurrence and its instruction length. An appropriate metric is the quotient of the two as defined in equation 1 below. \( p(x) \) equals to 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.

\[
v(x) = \frac{p(x)}{w(x)} \quad \forall x \in X
\]

Our goal is to design a \( n/2n \)-bit instruction set consisting of single-worded \( n \)-bit instructions and double-worded \( 2n \)-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 that requires \( 2^{h(x)+*} \) bits of the \( 2^{2n} \) encoding space and a double-word instance that requires only \( 2^{h(x)} \) bits. We will call this total set of instruction candidates \textit{intermediate representation}. Figure 3 shows an example of a 4/8-bit input instruction set and its intermediate representation. \( X_8 \) represents an instruction of length \( b \) bits. Single-word instructions are printed as \( \bar{X}_8 \). The original instruction set consists of eleven instructions \( A \rightarrow N \) – three of which are single-worded (\( A, B \) 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 \( G_3 \)).

![Figure 3. original and intermediate instruction set](image)

The greedy algorithm uses an intermediate representation as 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.

4.2 Refinement

In the previous example we considered only a reduced version of our original problem. We did not create new variants of instructions that implement a subset of an original instruction’s functionality. The concept 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 \texttt{MOV Rx, imm(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 \texttt{MOV Rx, imm(3)} which needs to be considered when building an optimized instruction set. In the following text we will...
name such shorter instruction instances \( X^* \) and their longer
variant \( X_B(N) \) where \( B > b \) and \( N > n \). Whenever \( X^*_i \) 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 \( x^*_i \) that are already selected.
On the other hand if \( x \) enters the selection all previously selected
shorter instructions \( x^*_i \) leave the selection instantly. So the actual
codespace-requirement of selecting \( x \) must be reduced by the sum
of all its already selected subinstructions.

\[
v(x) = \frac{p(x) - \sum_i p(x^*_i)}{c(x) - \sum_j c(x^*_j)} \quad (2)
\]

Likewise we have to zero the value of all instructions \( X^*_i \) 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.

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

Figure 4 shows how to use the modified value function
described above. We simply introduced a subinstruction \( G^*_i \) which
would be sufficient for 15 instructions of the input program. Note
that those 15 instances are already included in \( G^3 \)'s count of 21.
Therefore the input program's total number of instructions (94) and
the code size (716) did not change.

The algorithm starts again by selecting all double-word instruc-
tions. Afterwards the most valuable instruction is checked whether
it fits into the remaining codespace. The most valuable instruction
is our subinstruction \( G^*_i \) with a value of 0.234. We select it and
recalculate the value of all its superinstructions as defined by equa-
tion 2. This changes \( G^*_i \)'s value from 0.164 to 0.094. Also note that
different from our previous example we do not remove \( G^*_i \) as only
15 out of its 21 instructions are covered by \( G^*_i \). The remaining 6 in-
structions still need \( G^3 \)'s instruction instance. The next two instruc-
tions selected are \( A_5 \) and \( E_0 \). At the end only 8 bits of codespace
are available which can’t be filled up by any of the remaining can-
didates – so we are finished. Recoding the input program with our
new optimized instruction set (including the new instruction \( G^* \)
results in a smaller program of only 644 bits.

We are now able to give the pseudocode–listing (see figure 5 of
the greedy algorithm. The value function \( v(x) \) referenced therein is
defined as in equation 2.

4.3 Discussion
As we will see in chapter 6 the greedy algorithm produces quite
good but not always optimal results. Such a case is outlined in
figure 6. It is similar to the example used above but we added 4
additional \( E \)-instructions (and removed subinstruction \( G^* \) to keep
the example simpler). The algorithm runs as usual and finds a
solution that requires 708 bits. The optimal solution which can be
calculated by algorithms we will introduce later on requires only
680 bits. The reason for that difference is the algorithm’s way of
selecting “the best” element. Selecting \( E_0 \) at step 1 requires 60
bits of the remaining 162 bits of remaining codespace. The next
most valuable element \( G^*_2 \) then does not fit into the remaining 102
bits as it would require \( \text{120 bits} \). But it would have been a better
solution if the algorithm had selected \( G^*_2 \) instead of \( E_0 \). This is
due to the following fact: 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 best advantage of our greedy algorithm is its runtime.
In contrast to all the other algorithms discussed later the greedy
algorithm is amazingly fast. Theoretically 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 6) show a difference
of less than 1 percent to the optimum.

4.4 Optimizations
The pseudocode of listing 5 recalculated all values and completely
resorted the set \( S \) each time an element was selected. But an ele-
ment’s value and its rank change only when one of its subelements
had just entered the result set. Therefore we can keep all other ele-
ments’ values and recalculate just the affected elements. The opti-
mized pseudocode is listed in figure 7.

5. ILP-based Algorithm
This section describes how to transform the instruction set opti-
mization problem into an integer linear programming (ILP) formu-
lation which can be solved using standard available software like
lpsolve.

5.1 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
instruction used by the input program. This goal is expressed in
equation (4). For each original instruction we introduce a binary
variable \( x_i \) that is 1 if the instruction is encoded as a single-word
and 0 otherwise. The \( p_i \)'s are constants counting how often an
instruction occurred in the input program. Equation (5) expresses

\[
S = \text{intermediate representation including counts}
\]

\[
c_s = 2^n \ldots n/2^n\text{-bit code requested}
\]

\[
R = \{ x | x \in S \wedge x \text{ is doubleword} \}
\]

\[
S = S \setminus R
\]

\[
\text{WHILE} \quad c_s < 0 \land S \neq \emptyset
\]

\[
\text{DO}
\]

\[
\text{SORT} \ S \text{ on } v(x) \text{ FOR ALL } x \in S
\]

\[
x = \text{top}(S)
\]

\[
\text{WHILE} \quad S \neq \emptyset \land \text{size}(x) > c_s
\]

\[
\text{DO}
\]

\[
R = R \cup \{ x \}
\]

\[
S = S \setminus x
\]

\[
c_s = c_s - \text{space}(x)
\]

\[
\text{RETURN} \ c_s
\]

Figure 5. Pseudocode of greedy algorithm
S = intermediate representation including counts
\( c_S = 2^{2n} \ldots n/2 \text{-bit code requested} \)
\( R = \{ x \mid x \in S \land x \) (doubleword) \}
\( S = S \setminus R \)
SORT S on \( v(x) \) FOR ALL \( x \in S \)
WHILE \( c_S < 0 \) AND \( S \neq \emptyset \)
DO
REPEAT
\( x = \text{top}(S) \)
\( S = S \setminus \{ x \} \)
UNTIL \( S = \emptyset \) OR \( \text{size}(x) \leq c_S \)
\( R = R \cup \{ x \} \)
\( S = S \setminus \text{sub}(x) \)
\( c_S = c_S - \text{space}(x) \)
FOR ALL \( y \mid x \in \text{sub}(y) \)
REARRANGE \( y \) in \( S \) on new \( v(y, x) \)
DONE
RETURN \( R \)
FUNCTION \( v(y, x) \)
\( \text{space}(y) = \text{space}(y) - \text{space}(x) \)
\( \text{count}(y) = \text{count}(y) - \text{count}(x) \)
RETURN \( \text{count}(y)/\text{space}(y) \)

Figure 7. Pseudocode of greedy algorithm – optimized

For the definition of our goal function we have to introduce new variables because we can’t use \( y_{j,k} \) from above. We create a new variable \( y_{j,k} \) and a corresponding constant \( p_{j,k} \). \( y_{j,k} \) is 1 if the instruction or some more general instruction is single-word encoded single-word (8). We use the notation \( a \supset b \) if an instruction variant \( a \) is more general than \( b \). If \( a = 0 \) this it 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 up to the number of original instruction instances (9). With these new constants and variables we can define our goal function as in (10).

\[
y_{j,k} = \begin{cases} 1 & \text{if } \exists s \supset k : \exists j,s = 1 \\ 0 & \text{otherwise} \end{cases}
\]

\[
p_{j,k} = \sum_{k=0}^{m} p_{j,k} = p_j
\]

\[
\text{maximize } \sum_{j=1}^{n} p_j x_j
\]

\[
\sum_{j=1}^{n} w_j x_j + w_j (1 - x_j) \leq c_j,
\]

\[
0 \leq x_j \leq 1.
\]

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.

5.2 Enhanced ILP maximization problem

For each new specialized instruction we define a new binary vari-able \( x_{j,k}, k \geq 1 \). Original instructions are represented by the binary variable \( x_{j,0} \). Whenever an instruction is encoded as a single-word the corresponding variable is 1 and 0 otherwise. Constants \( m_{j,k}, k \geq 0 \) hold the codespace that is required when encoding an instruction as a single-word. We already know that it does not make sense to create 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 which is represented by \( w_j \). Using these constants and the binary variables \( x_{j,k} \) we are able to redefine our codespace constraint as in (6).

\[
\sum_{j=1}^{n} m_{j,k} (1 - x_{j,0}) + \sum_{j=1}^{n} \sum_{k=0}^{m} w_{j,k} x_{j,k} \leq c_k
\]

\[
x_{j,k} = \begin{cases} 1 & \text{if single-word encoded} \\ 0 & \text{otherwise} \end{cases}
\]

For the definition of our goal function we have to introduce new variables because we can’t use \( y_{j,k} \) from above. We create a new variable \( y_{j,k} \) and a corresponding constant \( p_{j,k} \). \( y_{j,k} \) is 1 if the instruction or some more general instruction is single-word encoded single-word (8). We use the notation \( a \supset b \) if an instruction variant \( a \) is more general than \( b \). If \( a = 0 \) this it 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 up to the number of original instruction instances (9). With these new constants and variables we can define our goal function as in (10).

\[
y_{j,k} = \begin{cases} 1 & \text{if } \exists s \supset k : \exists j,s = 1 \\ 0 & \text{otherwise} \end{cases}
\]

\[
p_{j,k} = \sum_{k=0}^{m} p_{j,k} = p_j
\]

\[
\text{maximize } \sum_{j=1}^{n} p_j y_{j,k}
\]

In order to complete our ILP model we need to define some additional constraints that express the relations between the in-struction 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 \( y_{j,k} \) : whenever a more general instruction is counted as a single-word instruction then cer-tainly 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.

\[
\forall a \supset b : \quad x_{j,a} \Rightarrow - x_{j,b} \Leftrightarrow x_{j,b} + x_{j,a} \leq 1
\]

\[
\forall a \supset b : \quad y_{j,a} \Rightarrow y_{j,b} \Leftrightarrow y_{j,b} - y_{j,a} \geq 0
\]

\[
\forall a \supset b : \quad y_{j,a} \Rightarrow - y_{j,b} \Leftrightarrow y_{j,a} - y_{j,b} \geq 0
\]

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).

\[
\forall b : \quad y_{j,b} \Rightarrow x_{j,a} \vee (\exists a \supset b : \exists j,a = 1) \Leftrightarrow
\]

\[
\left( y_{j,b} + \sum_{a \supset b} y_{j,a} \right) - y_{j,b} \geq 0
\]

5.3 ILP problem solvers

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

- Ip_solve is a MIP (mixed integer programming) solver. Ip_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 (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 writing a heuristic on 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 solved our problem instances within a few seconds. Unexpectedly the runtime of OPBDP which is specialized for nonlinear 0-1 optimization problems was much higher.

5.4 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-worded ADD Rx, imm(3) instruction whenever a SUB Rx, imm(3) is encoded as a single-word. With our ILP model we only need to add a simple equation representing that constraint: $\sum_{ADD,3} - \sum_{SUB,3}$.

6. Experimental Results
Our work is based on an existing instruction set that was designed by experienced architects over some time. 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 percent of the 40-bit instructions require less 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 less than 50 percent of all the available instructions are ever used. This alone theoretically opens high optimization potential when we are allowed to create a reduced instruction set containing only those instructions that are actually used. But only few of the previous 51 40-bit instructions requiring less than 20 bits are used by our benchmark programs. So the potential is practically rather low.

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 1 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 1 we learn that 20 up to 50 percent of our programs’ code size is made up by double-word instructions. At best we can therefore theoretically achieve up to 25 percent 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.

6.1 Locally optimized instruction sets
This section covers results when constructing an optimized instruction set for the DSP-benchmark programs introduced in the previous section.

Table 2 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 new 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 6.2 we will present results for globally optimized instruction sets.

<table>
<thead>
<tr>
<th>benchmark program</th>
<th>original code size</th>
<th>greedy algorithm</th>
<th>exact algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>code size</td>
<td>%</td>
<td>code size</td>
</tr>
<tr>
<td>adpcm</td>
<td>13920</td>
<td>100.0</td>
<td>12800</td>
</tr>
<tr>
<td>bio/whirl</td>
<td>18880</td>
<td>100.0</td>
<td>15980</td>
</tr>
<tr>
<td>cmisc</td>
<td>89720</td>
<td>100.0</td>
<td>60360</td>
</tr>
<tr>
<td>dcc32</td>
<td>22880</td>
<td>100.0</td>
<td>20000</td>
</tr>
<tr>
<td>dct8x8</td>
<td>18740</td>
<td>100.0</td>
<td>16920</td>
</tr>
<tr>
<td>dot</td>
<td>10020</td>
<td>100.0</td>
<td>9280</td>
</tr>
<tr>
<td>gsm</td>
<td>96100</td>
<td>100.0</td>
<td>85700</td>
</tr>
<tr>
<td>g/21</td>
<td>54500</td>
<td>100.0</td>
<td>31300</td>
</tr>
<tr>
<td>ghs</td>
<td>59340</td>
<td>100.0</td>
<td>53980</td>
</tr>
<tr>
<td>rijndael</td>
<td>70300</td>
<td>100.0</td>
<td>63480</td>
</tr>
<tr>
<td>serpent</td>
<td>93000</td>
<td>100.0</td>
<td>85240</td>
</tr>
<tr>
<td>viterbi</td>
<td>22620</td>
<td>100.0</td>
<td>19380</td>
</tr>
<tr>
<td>synthetic</td>
<td>284540</td>
<td>100.0</td>
<td>258520</td>
</tr>
</tbody>
</table>

Table 2. benchmark analysis - 20/40 bit code locally optimized

Analysing the date of this figure demonstrates that for some programs the theoretical optimum – i.e. encoding all 40-bit instructions as single-word instructions – is nearly reached. For example 24.1 percent of the rijndael benchmark program’s code size was made up by 40-bit instructions. Theoretically any optimization technique that does not change important architectural features is limited to 12 percent – with the technique presented here we gained 10.2 percent.

Figure 8 shows for each benchmark application a comparison of the optimized 20/40-bit and 16/32-bit instruction set.

6.2 Globally optimized instruction sets
For the data presented in table 3 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 we can see from table 3 an instruction set optimized for one program can have 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 benchmark – with a code size increase of up to 3.5 percent.
The ILP based is fast enough to be used also for large problems. Evaluation showed that on average a given program's code size can be reduced by 11 percent. Some applications even allow for a reduction by up to 20 percent. This was achieved without changing fundamental architectural components of the underlying processor. Further optimization is possible when the bit-length of the instruction set is reduced. This yields an improvement of additional eight percent compared to the original code size.

6.3 Algorithm run time

We compared the run time of the greedy and ILP algorithm. This was interesting as theoretically 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 4 shows the time the two algorithms spent computing a single instruction for a reduction by up to 20 percent. This was achieved without changing fundamental architectural components of the underlying processor. Further optimization is possible when the bit-length of the instruction set is reduced. This yields an improvement of additional eight percent compared to the original code size.

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

Evaluation showed that on average a given program’s code size can be reduced by 11 percent. Some applications even allow for a reduction by up to 20 percent. This was achieved without having to change fundamental architectural components of the underlying processor. Further optimization is possible when the bit-length of the instruction set is reduced. This yields an improvement of additional eight percent compared to the original code size.

Table 1. benchmark programs - code size analysis

<table>
<thead>
<tr>
<th>benchmark program</th>
<th>code size[bits] 20-bit</th>
<th>code size[bits] 40-bit</th>
<th>code size [%] 20-bit</th>
<th>code size [%] 40-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>38</td>
<td>15</td>
<td>53</td>
<td>10080</td>
</tr>
<tr>
<td>blowfish</td>
<td>40</td>
<td>15</td>
<td>55</td>
<td>12480</td>
</tr>
<tr>
<td>hmac</td>
<td>53</td>
<td>23</td>
<td>76</td>
<td>40760</td>
</tr>
<tr>
<td>dct8x8</td>
<td>39</td>
<td>13</td>
<td>52</td>
<td>13640</td>
</tr>
<tr>
<td>dot</td>
<td>32</td>
<td>11</td>
<td>43</td>
<td>7460</td>
</tr>
<tr>
<td>g721</td>
<td>51</td>
<td>23</td>
<td>74</td>
<td>62740</td>
</tr>
<tr>
<td>gsm</td>
<td>51</td>
<td>23</td>
<td>74</td>
<td>62740</td>
</tr>
<tr>
<td>ct721</td>
<td>50</td>
<td>13</td>
<td>63</td>
<td>23600</td>
</tr>
<tr>
<td>ghs</td>
<td>43</td>
<td>18</td>
<td>61</td>
<td>43400</td>
</tr>
<tr>
<td>rijndael</td>
<td>43</td>
<td>18</td>
<td>61</td>
<td>43400</td>
</tr>
<tr>
<td>serpent</td>
<td>54</td>
<td>21</td>
<td>75</td>
<td>73160</td>
</tr>
<tr>
<td>viterbi</td>
<td>42</td>
<td>17</td>
<td>59</td>
<td>14140</td>
</tr>
<tr>
<td>synthetic</td>
<td>64</td>
<td>27</td>
<td>88</td>
<td>185860</td>
</tr>
</tbody>
</table>

Table 2. benchmark programs - code size analysis

<table>
<thead>
<tr>
<th>benchmark program</th>
<th>code size[bits] 20-bit</th>
<th>code size[bits] 40-bit</th>
<th>code size [%] 20-bit</th>
<th>code size [%] 40-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>84.9</td>
<td>89.1</td>
<td>92.4</td>
<td>10080</td>
</tr>
<tr>
<td>blowfish</td>
<td>92.2</td>
<td>90.7</td>
<td>85.1</td>
<td>12480</td>
</tr>
<tr>
<td>hmac</td>
<td>93.6</td>
<td>95.3</td>
<td>94.3</td>
<td>40760</td>
</tr>
<tr>
<td>dct8x8</td>
<td>97.2</td>
<td>96.2</td>
<td>96.6</td>
<td>13640</td>
</tr>
<tr>
<td>dot</td>
<td>96.5</td>
<td>94.4</td>
<td>93.9</td>
<td>7460</td>
</tr>
<tr>
<td>g721</td>
<td>97.6</td>
<td>95.6</td>
<td>96.5</td>
<td>62740</td>
</tr>
<tr>
<td>gsm</td>
<td>97.8</td>
<td>95.6</td>
<td>95.4</td>
<td>62740</td>
</tr>
<tr>
<td>ct721</td>
<td>97.5</td>
<td>95.5</td>
<td>94.8</td>
<td>23600</td>
</tr>
<tr>
<td>ghs</td>
<td>97.3</td>
<td>95.3</td>
<td>94.8</td>
<td>43400</td>
</tr>
<tr>
<td>rijndael</td>
<td>96.8</td>
<td>94.8</td>
<td>93.7</td>
<td>73160</td>
</tr>
<tr>
<td>serpent</td>
<td>96.9</td>
<td>94.9</td>
<td>93.8</td>
<td>14140</td>
</tr>
<tr>
<td>viterbi</td>
<td>97.1</td>
<td>95.0</td>
<td>94.0</td>
<td>185860</td>
</tr>
<tr>
<td>synthetic</td>
<td>97.9</td>
<td>96.1</td>
<td>94.8</td>
<td>185860</td>
</tr>
</tbody>
</table>

Table 3. benchmark analysis - 16/32 bit code globally optimized

<table>
<thead>
<tr>
<th>benchmark program</th>
<th>cputime [seconds]</th>
<th>greedy algorithm</th>
<th>ILP algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>0.07</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>blowfish</td>
<td>0.04</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>hmac</td>
<td>0.07</td>
<td>0.79</td>
<td></td>
</tr>
<tr>
<td>dct8x8</td>
<td>0.04</td>
<td>0.30</td>
<td></td>
</tr>
<tr>
<td>dot</td>
<td>0.02</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>g721</td>
<td>0.05</td>
<td>0.47</td>
<td></td>
</tr>
<tr>
<td>ghs</td>
<td>0.05</td>
<td>0.39</td>
<td></td>
</tr>
<tr>
<td>rijndael</td>
<td>0.05</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>serpent</td>
<td>0.06</td>
<td>1.27</td>
<td></td>
</tr>
<tr>
<td>synthetic</td>
<td>0.17</td>
<td>4.18</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. algorithm runtime for locally optimized 16-32 bit instruction set
Acknowledgments

References


Original n/2n instruction set (94 instr./716 bits)  

<table>
<thead>
<tr>
<th>bits</th>
<th>instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(N_5(12))</td>
</tr>
<tr>
<td>4</td>
<td>(M_4(25))</td>
</tr>
<tr>
<td>3</td>
<td>(G_3(21), H_3(9), I_3(8))</td>
</tr>
<tr>
<td>2</td>
<td>(B_2(3), C_2(1), D_2(2), E_2(7), F_2(1), G_2^*(7))</td>
</tr>
<tr>
<td>1</td>
<td>(A_1(5))</td>
</tr>
</tbody>
</table>

Intermediate 2n representation (count + value) :
7  \(G_7(21)[0.164], H_7(9)[0.070], I_7(8)[0.063]\)
6  \(M_6(3)[0.047], C_6(1)[0.016], D_6(2)[0.031], E_6(7)[0.109], F_6(1)[0.016], G_6^*(15)[0.23]\)
5  \(A_5(5)[0.156], N_5(12)\)
4  \(M_4(25)\)
3  \(G_3(21), H_3(9), I_3(8)\)
2  \(B_2(3), C_2(1), D_2(2), E_2(7), F_2(1)\)
1  \(A_1(5)\)

Algorithm steps :  

<table>
<thead>
<tr>
<th>step instructions</th>
<th>remaining codespace [bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, E_2, F_2, A_1)</td>
<td>162</td>
</tr>
<tr>
<td>1  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, E_2, F_2, A_1, G_6^*)</td>
<td>98 = 162 - 2^6</td>
</tr>
<tr>
<td>2  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, E_2, F_2, G_6^*, A_6)</td>
<td>68 = 98 - 2^3 + 2^1</td>
</tr>
<tr>
<td>3  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, F_2, G_6^*, A_6, E_6)</td>
<td>8 = 68 - 2^6 + 2^2</td>
</tr>
</tbody>
</table>

Result : 94 instr./644 bits (10.1% improvement)

\(^*G_7's\) value is now recalculated to \(\frac{13}{125 - \frac{15}{20}} = 0.094\); also note that \(G_3\) stays selected

---

Original n/2n instruction set (98 instr./748 bits)  

<table>
<thead>
<tr>
<th>bits</th>
<th>instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(N_5(12))</td>
</tr>
<tr>
<td>4</td>
<td>(M_4(25))</td>
</tr>
<tr>
<td>3</td>
<td>(G_3(21), H_3(9), I_3(8))</td>
</tr>
<tr>
<td>2</td>
<td>(B_2(3), C_2(1), D_2(2), E_2(11), F_2(1))</td>
</tr>
<tr>
<td>1</td>
<td>(A_1(5))</td>
</tr>
</tbody>
</table>

Intermediate 2n representation (count + value) :
7  \(G_7(21)[0.164], H_7(9)[0.070], I_7(8)[0.063]\)
6  \(M_6(3)[0.047], C_6(1)[0.016], D_6(2)[0.031], E_6(11)[0.172], F_6(1)[0.016]\)
5  \(A_5(5)[0.031], N_5(12)\)
4  \(M_4(25)\)
3  \(G_3(21), H_3(9), I_3(8)\)
2  \(B_2(3), C_2(1), D_2(2), E_2(11), F_2(1)\)
1  \(A_1(5)\)

Algorithm steps :  

<table>
<thead>
<tr>
<th>step instructions</th>
<th>remaining codespace [bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, E_2, F_2, A_1)</td>
<td>162</td>
</tr>
<tr>
<td>1  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, E_2, F_2, A_1, F_6)</td>
<td>102 = 162 - 2^6 + 2^2</td>
</tr>
<tr>
<td>2  (N_5, M_4, G_3, H_3, I_3, B_2, C_2, D_2, F_2, G_6, A_6)</td>
<td>72 = 102 - 2^5 + 2^1</td>
</tr>
<tr>
<td>3  (N_5, M_4, G_3, H_3, I_3, C_2, D_2, F_2, G_6, A_6, E_6)</td>
<td>12 = 72 - 2^3 + 2^2</td>
</tr>
</tbody>
</table>

Result : 98 instr./708 bits

Optimum : 98 instr./680 bits

\(N_5, M_4, H_3, I_3, B_2, C_2, D_2, E_2, F_2, G_7, A_6\)

---

**Figure 4.** greedy algorithm - modified value function

---

**Figure 6.** greedy algorithm - non-optimal solution