# Summary of the paper "Compiling with Code-Size Constraints" [Naik and Palsberg, 2004]

Kenneth Rohde Christiansen

Department of Mathematics and Computing Science
Rijksuniversiteit Groningen
Blauwborgje 3
NL-9747 AC Groningen

k.r.christiansen@student.rug.nl / kenneth@gnu.org

**Abstract**
In this summary we discuss a paper presenting a way to compile a program with code-size constraints. The authors use Integer Linear Programming to accomplish this, so a short introduction will be given to this as well. The document assumes that the reader has followed the Embedded Systems course given at the RUG or at least a similar course given at another research institution.

**Keywords:** Code-Size Constraints, Embedded Systems, Integer Linear Programming, Performance, Computer Science.

## 1 Introduction

One of those things that makes it hard to develop for embedded systems is that there is the challenge of making the needed functionality fit into the available code space. The code space is often very limited since economical and competitive considerations requires the companies to use cheap and small processors. Not only does the developed have to implement the required functionality, but at the same time the developer must do all kinds of tricks to make it fit. This often leads to very complicated and hardly maintainable code.

When developing the application the developer can choose between coding the project directly in assembler language or in a higher level language, which will most often be something similar to the C programming language.

Coding in assembler is not easy and the code is a lot harder to understand, debug and maintain. Coding in assembler, though, makes it possible to easier optimize for code size. On the other hand, coding in a high level language makes reuse, maintainability, portability and programming a lot easier. Unfortunately, there is not much control of the code. The idea behind high level languages is also that you do not need to worry about these things, as the compiler will take care of it for you. The problem is, though, that most compilers emphasis on execution speed. Many of the optimization techniques tend to make the code size larger for instance by unrolling loops and inlining functions.

The question is whether it is possible to compile a high level language so that the code is optimized for code-size and the developed doesn't have to worry about this. Thus, it is possible to get the best of both worlds.

The paper shows that there has been quite some research in this area and it is indeed possible to optimize for code-size by various compiler techniques. In [Park and Moon, 2001] register allocation for an architecture with two symmetric banks is studied. The reviewed paper presents three additional techniques for optimizing code-size. They handle interrupts, do whole-program register allocation and enable saving the register pointer on the stack, which is useful when the interrupt handler is invoked.

## 1.1 The target architecture

The paper discusses compilation for the Z86E30 processor from Zilog. According to the paper, this is a widely used microprocessor. The processor uses a banked register architecture, which means that the techniques presented in the paper can be adapted to similar architectures, like for instance the Intel 8051.

The high level language used in the paper is called ZIL. It is a quite simple language that resembles C in some respects. The language supports interrupts and also includes some low-level instructions for handling these. The compilation will be from ZIP to the Z86 assembler language. An overview over the language can be found in [Naik and Palsberg, 2004]

### 1.1.1 Main problems

The main problem with compiling for the Z86E30 architecture is that the architecture has no stack. Every variable must then be store in registers. Another problem is distributing the variables in the register banks. The reason for this, is that some assembler instructions can be considered RP (register pointer) sensitive. Lets explain this a bit further.

Each Z86 assembler instruction can address a register with 4 or 8 bits. If 8 bits is used the first 4 bits represent the bank number and the last 4 bits represent the register number within this bank. If 4 bits are used, the bank number that the RP is pointing to, is used.

As a consequence of this, some of the instructions require an extra byte when the RP is not pointing at the bank which contains the variables in use in the instruction. Some of these are: **inc** $v$ and **add** $v_1$, $v_2$

According to the authors, 30% of the instructions of their benchmark programs were RP-sensitive, so good register allocation was their main target.

## 1.2 0-1 Integer Linear Programming

The authors implemented a ZIL to Z86 compiler with the use of Integer Linear Programming. A linear program ( [Shenoy, 1989]) is a problem that can be expressed in the so-called Standard Form:

minimize $cx$ subject to $Ax = b, x >= 0$

Here $x$ is the vector of variables to be solved for, $A$ is a matrix of known coefficients, and $c$ and $b$ are vectors of known coefficients. The expressing $cx$ is called an objective function and $Ax = b$ are the so-called constraints. Since A is seldom square, it is very hard solving these equations and it is in fact an NP-complete problem and the problem can be converted to any other NP-complete problem, as for instance the problem of the Traveling Salesman that everyone knows so well.

Integer linear programming (ILP) requires all (some, depending on the definition) of the variables to take integer values. What might not seem so obvious is that solving ILP is in fact a lot harder than solving non-integer linear programming problems.

The 0-1 in 0-1 Integer Linear Programming means that the integers can only take the values 0 or 1, thus you can consider it some kind of Binary/Boolean Linear Programming.

## 1.3 Procedure

The procedure followed by the authors was to first extract the model of the ZIL program, in order to get a control-flow graph. After this a so-called AMPL tool was used to generate the actual ILP problem from the control-flow graph and the ILP formation. This tool is discussed in [Fourer and Kernighan, 1993]. In order to solve the 0-1 ILP problem, a commercial solver from ILOG (http://www.ilog.com) was used. A code generator was then used to generate the actual Z86 code from the solution given by the CPLEX ILP Solver from ILOG. If the reader is interested, a description of the code generation procedure can be found in the summarized paper.

### 1.3.1 The ILP Formulations

In the paper three ILP formulations were used. The reason for this is, as said before, that solving ILP requires NP-complete algorithms. The simpler ILP formulation used, the quicker a solution to the problem is found.

The used formulations are:

- **Inexpensive:** It is only allowed to set the register pointer (with the instruction `srp`) at the beginning at the program
- **Selective:** The register pointer can be set at the entry and the exit points of procedures and interrupt handlers. Additionally, the register pointer can be pushed at the entry of interrupt handlers and popped at the exit point.
- **Exhaustive:** The register pointer can be set before any instruction, but pushing (and popping) the register pointer is restricted to the entry (exit resp.) point of the interrupt handlers.

According to the paper, **Selective** proved to be a good candidate, since it offered a good trade-off between the solving time of the ILP and the saving of code-size.

### 1.3.2 The Inexpensive Formulation

Since this is just a summary we won't present all of the actual formulations. All of these can be found in the original paper and we suggest the interested reader to take a look at these. In order to show what such a formulation can look like, we will describe the most simple formulation in this section.

In the formulation we use the following definitions:

- `Var` denotes all variables in the program.
- Subsets $\texttt{PDV0} \in \texttt{Var}$ and $\texttt{PDV15} \in \texttt{Var}$ are subsets of predefined variables that have been allotted to be in register 0 and 15 respectively.
- $\texttt{Bin}X\texttt{Inst}$ are RP-sensitive instructions that take $X = \{1, 2\}$ variables.
- `DjnzInst` are 'Decrement and jump if nonzero' instructions.

We first present the formulation and then comment on what it actually means:

**Objective function:** We want to minimize the following expressing:

$$\sum_{(i,v_1,v_2)\in\texttt{Bin2Instr}} \texttt{Bin2Cost}_i$$

$$- \sum_{(i,v)\in\texttt{Bin1Instr}} \texttt{InCurrBank}_v$$

**Linear constrains:** The minimization will be done subject to the following constrains:

1. $\sum_{v\in\texttt{Var}} \texttt{InCurrBank}_v \leq 16$. Only 16 variables can be stored in a bank $b$.
2. $\forall v_1 \in \texttt{PDV0}, \forall v_1 \in \texttt{PDV15} : \texttt{InCurrBank}_{v_1} + \texttt{InCurrBank}_{v_2} = 1$. This states that variables in `PDV0` and `PDV15` cannot simultaneously be in bank $b$.
3. $\forall v_1 \in \texttt{PDV0}, \forall v_1 \in \texttt{PDV0} : \texttt{InCurrBank}_{v_1} = \texttt{InCurrBank}_{v_2}$. Variables in `PDV0` have to simultaneously be in bank $b$ or in a different bank.
4. $\forall v_1 \in \texttt{PDV15}, \forall v_1 \in \texttt{PDV15} : \texttt{InCurrBank}_{v_1} = \texttt{InCurrBank}_{v_2}$. Same as above, but for `PDV15`.
5. $\forall (i, v) \in \texttt{DjnzInstr} : \texttt{InCurrBank}_v = 1$. All operants of a `djnz` instruction must be stored in the same bank $b$.
6. $\forall (i, v_1, v_2) \in \texttt{Bin2Instr} : \texttt{Bin2Cost}_i + \texttt{InCurrBank}_{v_1} \geq 1$. States that $(i, v_1, v_2) \in \texttt{Bin2Cost}_i$ is 1 if $v_1$ is stored in a different bank than $b$.
7. $\forall (i, v_1, v_2) \in \texttt{Bin2Instr} : \texttt{Bin2Cost}_i + \texttt{InCurrBank}_{v_2} \geq 1$. Same as above, but for $v_2$.

### 1.3.3 Model extraction

In order to solve the ILP problem we need a control flow graph as well as an ILP formulation. The model extraction is done in the following manner:

- Extra `skip` instructions are added at the entry points of each procedure, in order to make sure that there are no jumps to the instructions at the start of procedures.
- For each program point the value of the Interrupt Mask Register (IMR) is estimated. This means that the interrupt handlers that are enables for each program point are estimated. A special technique is used for accomplishing this. This technique is described in [Brylow and Palsberg, 2001].

- The actual Control-Flow graph is build with all occurances of instructions $i_x$ as nodes in a directed graph with edges ($i_1$, $i_2$), where $i_2$ is executed immediate after $i_1$ in a program run. There will also be edges corresponding to the interrupt handlers and returns as estimated by the IMR estimate.
- Since we are not interested in register-pointer-insensitive instructions we extract a subgraph from the Control-Flow graph. We do that by only keeping register-pointer-sensitive instructions as well as `ret` (return), `iret` (interrupt return), `djnz` (decrement and jump if non-zero) as well as `skip` instructions at the entry point of procedures (those we inserted ourselves).

An example of a Control-Flow graph can be seen in the original paper.

#### 1.3.4 Final words

We have decided to leave out code generation from this summary since it is a bit technical and tied to the Z68 assembler language. The method can be found in the summarized pages, as well as an example of the generated code for some example program proceeded with the **inexpensive** formulation. We suggest the interested reader to take a look at this, as well as some of the of the other formulations.

## 2 Results

In order to evaluate the method used, the authors got hold of six hand-written Z86 assembler programs written by Greenhill Manufacturing; a company with great experience with the Z86 assembler language. A reverse engineering tool was used to convert the applications to ZIL. Since ZIL doesn't support mutable arrays and assembler tricks, like jumping from one routine to instructions in another routine (something that doesn't help maintainability and reuse) it is not possible to reach as little a code-size as with the hand-written programs.

The two ILP formulations, **inexpensive** and **exhaustive** were evaluated by comparing ILP solving time with the code-size reduction. It turned out that the solving time for **exhaustive** was very high and the authors looked at what could be done for reducing this. By looking at where the `srp` (set register pointer) was inserted, the **selective** formulation was invented and it proved to be a good trade-off between the two other ILP formulations. It turns out that **exhaustive** only saves 0.5% of the code-size in comparison with **selective**, where as the solving time is a lot greater. The actual numbers can be found in the summarized paper.

## 3 Conclusion

The authors of the paper looked into what could be done in order to compile ZIL source code to Z86 assembler with code-size constrains. The architecture of the Z86E30 processor used has no stack and it turns out that distributing the variables in the register banks in the right fashion can reduce code-size, as some instructions are so-called RP-sensitive.

The authors used ILP in order to reduce code-size under certain constraints. Solving ILP requires NP-complete algorithms, so using the best possible ILP formulation might not be preferable. The authors showed that it was possible to develop a **selective** formulation that created almost as compact assembler code as the best possible formulation, **exhaustive** - and did so in a lot less time.

The authors proved that it indeed is possible developing compilers that can be used for compiling with code-size constraints. This is great news for developers working with embedded systems, as they can get all the advantages of using high-level programming languages without worrying about code-size constraints.

## 4 Acknowledgements

We want to thank our teacher Sietse Achterop for an interesting and worthwhile course.

## References

Brylow, D., D. M. and Palsberg, J. (2001). Static checking of interrupt-driven software. *Proceeding of ICSE'01, 23rd International Conference of Software Engineering.*, 47–56.

Fourer, R., G. D. and Kernighan, B. (1993).

*AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press.

Naik, M. and Palsberg, J. (2004). Compiling with code-size constraints. *ACM Transactions on Embedded Computing Systems*, **3**(1), 163–181.

Park, J., L. J. and Moon, S. (2001). Register allocation for banked register file. *Proceeding of LCTES'02, Languages, Compilers and Tools for Embedded Systems.*, 39–47.

Shenoy, G. V. (1989). *Linear Programming: Methods and Applications*. Halsted Pr.