Compiling with Code-Size Constraints

Mayur Naik
Purdue University
Department of Computer Science
West Lafayette, IN 47907
mnaik@cs.purdue.edu

Jens Palsberg
Purdue University
Department of Computer Science
West Lafayette, IN 47907
palsberg@cs.purdue.edu

ABSTRACT

Most compilers ignore the problems of limited code space in embedded systems. Designers of embedded software often have no better alternative than to manually reduce the size of the source code or even the compiled code. Besides being tedious and error-prone, such optimization results in obfuscated code which is difficult to maintain and reuse. In this paper, we present a code-size-directed compiler. We phrase register allocation and code generation as an integer linear programming problem where the upper bound on the code size can simply be expressed as an additional constraint. Our experiments show that our compiler, when applied to two commercial microcontroller programs, generates code as compact as carefully crafted code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms

Algorithms, Measurement, Performance

Keywords

register allocation, integer linear programming, space optimization, banked architecture

1. INTRODUCTION

1.1 Background

In an embedded system, it can be challenging to fit the needed functionality into the available code space. Economic considerations often dictate the use of a small and cheap processor, while demands for functionality can lead to a need for considerable code space. Thus, the designer of the software must both implement the desired functionality *and* do it with a limited code-space budget. There are at least two options for handling such a task:

- Write the software in assembly language; this gives good control over code size but makes programming, maintenance, and reuse hard; or
- Write the software in a high-level language; this gives poor control over code size but makes programming, maintenance, and reuse easier.

The reason why the latter option gives poor control over code size is that most compilers ignore the problems of limited code space in embedded systems. Typically, a compiler places primary emphasis on the execution speed of the compiled code, and much less emphasis on the size of the compiled code. In some cases, optimizations for execution speed conflict with optimizations for code size. For example, loop unrolling tends to make code faster and bigger. Another example is the use of procedures which tends to make code slower and smaller. In this paper, we focus on combining programming in high-level languages with control over code size.

> **Question: can we get the best of both worlds?** Can we get both the *flexibility* of programming in a high-level language and the *control over code size* that is possible when programming in assembly?

This question has been studied in the past decade by many researchers who have shown that good data layout can lead to reduced code size, see Figure 1. For example, in a seminal paper, Liao, Devadas, Keutzer, Tjiang, and Wang [10] demonstrated that on many contemporary digital signal processors, good data layout increases opportunities for using autoincrement/autodecrement addressing modes which in turn reduces code size significantly. In this paper, we present a code-size-directed compiler that generates code as compact as carefully crafted code on architectures in which the register file is partitioned into banks, with a Register Pointer (RP) specifying the "working" bank.

Park, Lee, and Moon [13] have studied register allocation for an architecture with two symmetric banks. They perform per-basic-block register allocation in one bank and per-procedure register allocation in the other, and they generate instructions for moving data between the banks and for moving RP. In this paper, we present three new techniques, namely, we handle interrupts; we do whole-program register allocation; and we enable saving RP on the stack by generating instructions such as:

Authors	Architecture	Good data layout increases the opportunities for using:
• Liao, Devadas, Keutzer, Tjiang, and Wang [10] • Leupers and Marwedel [9] • Rao and Pande [14]	contemporary digital signal processor	autoincrement/autodecrement addressing modes
• Sudarsanam and Malik [18]	two memory units	parallel data access modes
• Sjödin and von Platen [16]	multiple address spaces	pointer addressing modes
• Park, Lee, and Moon [13]	two register banks	RP-relative addressing
• this paper	multiple register banks	RP-relative addressing

Figure 1: Good data layout can significantly reduce code size

```
push RP
srp b        // set RP to bank b
 ...
pop RP
```

The idea of saving RP on the stack is particularly useful when an interrupt handler is invoked, but can also be useful when a procedure is called from several call sites.

1.2 The Problem: Compiling for the Z86E30

We address the problem of code-size-directed compilation on Zilog's Z86E30 processor [3] from ZIL to Z86 assembly language. ZIL is an intermediate language we have designed that strongly resembles Z86 assembly language except that it uses variables instead of registers. The grammar of ZIL along with some notes on the semantics and an example program is presented in Appendix A.

We have access to two proprietary microcontroller programs that were carefully handwritten in Z86 assembly language. We have assessed the quality of our code-size-directed compilation by rewriting these programs in ZIL, compiling them back to Z86 assembly language using our code-size-directed compiler, and comparing the size of the generated code to that of the original handwritten code.

The Z86E30 has 256 8-bit registers organized into 16 banks of 16 registers each. Of these, 236 are general-purpose, while the rest, namely, the first 4 registers in the 0^{th} bank and the 16 registers in the 15^{th} bank, are special-purpose. All special-purpose registers in the 0^{th} bank and 12 of the 16 special-purpose registers in the 15^{th} bank are visible to the ZIL programmer. The RP is itself a special-purpose register in the 15^{th} bank that is invisible at the ZIL level.

The Z86E30 lacks a data/stack memory; all variables must be stored in registers. As a result, whole-program register allocation must be performed, as opposed to per-procedure register allocation. Moreover, distributing global variables among the various banks in a manner that reduces the space cost of the whole program is a major challenge.

A Z86 assembly instruction can address a register using an 8-bit or 4-bit address. In the former case, the high nibble of the 8-bit address represents the bank number and the low nibble represents the register number within that bank. In the latter case, RP represents the bank number and the 4-bit address represents the register number within that bank. We shall refer to registers addressed using 4 and 8 bits as working and non-working registers respectively.

The space cost of certain Z86 assembly instructions depends upon whether they address registers using 4 or 8 bits. For instance, the cost of the unary decrement instruction **dec** v or the binary add instruction **add** v, c (c is a constant) is independent of the value of RP. But the cost of the unary increment instruction **inc** v or the binary add instruction **add** v_1, v_2 depends upon the value of RP: the cost of the former is 1 or 2 bytes depending on whether RP points to the bank in which v is stored or not, while the cost of the latter is 2 or 3 bytes depending on whether RP points to the bank in which v_1 and v_2 are stored or not.

Nearly 30-40% of the instructions in our benchmark programs occupy one fewer byte if they address registers using 4 bits. Thus, the key to optimizing the size of the target code generated for a ZIL program is optimal register allocation of the variables that are referenced by such instructions in the ZIL program.

1.3 Our Results

We have designed and implemented a code-size-directed compiler from ZIL to Z86 assembly language. We phrase register allocation as an integer linear programming (ILP) problem whose objective is to minimize the size of the target code. Our experiments show that our compiler, when applied to the two microcontroller programs, generates code that is as compact as the original handwritten code.

Our fully-automated approach is depicted in Figure 2. For a given ZIL program, we first perform model extraction to derive a control-flow graph. We then use the AMPL tool [6] to generate an ILP from the control-flow graph and an ILP formulation. Every variable in the ILP ranges over $\{0, 1\}$. The ILP is solved using an off-the-shelf ILP solver like CPLEX [1]. Once a solution has been obtained, a code generator produces a target Z86 program.

Our ILP formulation allows introducing **srp** immediately before any instruction in the model extracted from the ZIL program, and **push** (**pop**) **RP** immediately before the entry (exit) point of any procedure and interrupt handler. (Throughout the paper, when we speak of introducing an RP-manipulating instruction i immediately before an instruction i', we assume that any label at i' is moved to i.) We designate this ILP formulation SetRP + Full PuPoRP. We have experimented with three other ILP formulations, namely, SetRP + PuPoRP that allows introducing **push** (**pop**) **RP** for interrupt handlers but not for procedures, SetRP that does not allow introducing **push** (**pop**) **RP** at all, and Cheap that allows introducing just one **srp** at the start of the program. Our experiments show that SetRP + PuPoRP offers the best tradeoff between ILP solution time and code-space savings.

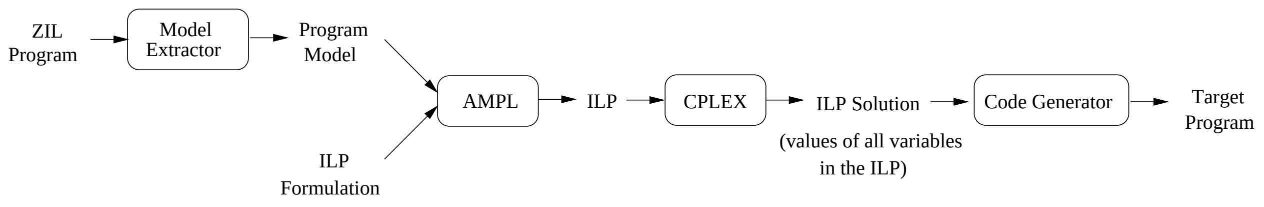


Figure 2: Overview

1.4 Related Work on ILP-based Compilation

In the past decade, there has been widespread interest in using ILP for compiler optimizations such as instruction scheduling, software pipelining, data layout, and, particularly, register allocation.

Goodwin and Wilken [7] pioneered the use of ILP for register allocation. Their approach is applicable only to processors with uniform register architectures. Kong and Wilken [8] present an ILP framework for irregular register architectures (architectures that place restrictions on register usage). Appel and George [4] partition the register allocation problem for the Pentium into two subproblems: optimal placement of spill code followed by optimal register coalescing; they use ILP for the former. Stoutchinin [17] and Ruttenberg et al. [15] present an ILP framework for integrated register allocation and software pipelining in the MIPS R8000 microprocessor. Liberatore et al.[11] perform local register allocation (LRA) using ILP. Their analysis shows that their approach is superior to a dynamic programming algorithm that solves LRA exactly but takes exponential time and space, as well as to heuristics that are fast but sub-optimal.

ILP has been widely used to perform register allocation for general-purpose programs on stock microprocessors but we are not aware of any ILP-based technique that has been used to perform register allocation for interrupt-driven programs on embedded microprocessors. The only ILP-based memory allocation techniques for embedded processors we are aware of are Sjödin and Platen's technique [16] which models multiple address spaces of a certain processor using ILP, and that of Avissar, Barua, and Stewart [12] which uses ILP to optimally allocate global and stack data among different heterogeneous memory modules.

1.5 Rest of the Paper

In the following section, we explain model extraction. In Section 3, we present our ILP formulation, namely, SetRP + Full PuPoRP. In Section 4, we discusse code generation. In Section 5, we present alternate ILP formulations. In Section 6, we show our experimental results and, finally, in Section 7, we conclude with a note on future work.

2. MODEL EXTRACTION

Model extraction proceeds in four steps.

First, we add an unlabeled **skip** instruction at the entry point of each routine to ensure that there are no jumps to it. The one introduced at the entry of `MAIN` is called i_{root}.

Secondly, we conservatively estimate at each instruction in the program, which interrupt handlers are enabled, *i.e.*, the value of the interrupt mask register (IMR). We use a technique closely related to the one described in [5].

Third, we construct a control-flow graph, denoted CFG, of the program. Let `Instr` be the set of all occurrences of instructions in the program. CFG is a directed graph whose set of nodes is `Instr` and each edge (i_1, i_2) is such that i_2 can be executed immediately after executing i_1 in a program run. Note that there will be edges corresponding to interrupt handler invocations and returns, as determined by the IMR estimates.

Finally, we build an abstraction of CFG, denoted CFG_{abs}, by eliminating those nodes and edges in CFG that are irrelevant to register allocation. Let $\texttt{Instr}_{abs}$ be the subset of `Instr` such that each instruction in the former satisfies at least one of the following two conditions:

- The instruction is a **ret**, **iret**, **call**, an instruction immediately following a **call**, a **skip** at the entry point of a routine, or **djnz**.
- The space cost of the instruction depends upon the value of RP, namely, upon whether RP points to the bank in which its operand(s) are stored or not.

CFG_{abs} is a directed graph whose set of nodes is $\texttt{Instr}_{abs}$ and each edge (i_1, i_2) is such that i_2 is reachable from i_1 in CFG along a path which does not contain any other node in $\texttt{Instr}_{abs}$, *i.e.*, i_2 can be executed after i_1 but before any other instruction in $\texttt{Instr}_{abs}$ in a run of the original program.

3. ILP FORMULATION

3.1 Set Declarations

- $\texttt{Bank} = \{0, \ldots, 12, 15\}$ is the set of banks in the register file of the Z86E30. Banks 13–14 are reserved for the run-time stack.
- $\texttt{Edge} \subseteq (\texttt{Instr}_{abs} \times \texttt{Instr}_{abs})$ is the set of edges in CFG_{abs}.
- $\texttt{ProcQuad} \subseteq (\texttt{Edge} \times \texttt{Edge})$ is the set of quadruples (i_1, i_2, i_3, i_4) such that i_2 and i_3 are the **skip** and **ret** instructions at the entry and exit of a procedure while edges (i_1, i_2) and (i_3, i_4) correspond to a call to the procedure and the corresponding return from it, respectively.
- `PuInstr` is the set of the **skip** instructions at the entry point of each routine (except MAIN) and `PoInstr` is the set of the **ret/iret** instructions at the exit point of each routine.
- $\texttt{ProcPair} = \{ (i_2, i_3) \mid (i_1, i_2, i_3, i_4) \in \texttt{ProcQuad} \}$. `IntrPair` is defined analogously for interrupt handlers.

ILP Formulation	Seconds to solve ILP	Code Size
SetRP	0.43	32
SetRP + PuPoRP	10.57	29
SetRP + Full PuPoRP	132.21	29

Figure 3: Results for `example.zil`

- `CallPair` $= \{ (i_1, i_4) \mid (i_1, i_2, i_3, i_4) \in$ `ProcQuad` $\}$. Also, `EdgeOrCallPair` = `Edge` $\cup$ `CallPair`.
- `PRetEdge` $= \{ (i_3, i_4) \mid (i_1, i_2, i_3, i_4) \in$ `ProcQuad` $\}$. `IRetEdge` is defined analogously for interrupt handlers. Also, `MiscEdge` = `Edge` $-$ (`PRetEdge` $\cup$ `IRetEdge`).
- `Var` is the set of all variables in the ZIL program. We assume that variables that are local to different routines and have the same name are renamed to eliminate name clashes in `Var`.
- `PDV0` $\subseteq$ `Var` and `PDV15` $\subseteq$ `Var` are the sets of the predefined variables for which special-purpose registers have been alloted in the 0^{th} and 15^{th} bank, respectively. We also define `PDV` = `PDV0` $\cup$ `PDV15` as the set of all predefined variables and `UDV` = `Var` $-$ `PDV` as the set of all user-defined ones.
- `Bin2Instr` $\subseteq$ (`Instr`$_{abs}$ $\times$ `Var` $\times$ `Var`) is the set of triples (i, v_1, v_2) such that i is a binary instruction with source and destination operands v_1 and v_2, respectively. The cost of every binary instruction with both variable operands in Z86E30's instruction set depends on the value of RP.
- `Bin1Instr` $\subseteq$ (`Instr`$_{abs}$ $\times$ `Var`) is the set of pairs (i, v) such that (i) i is a binary instruction with the same source and destination operand v, and (ii) the cost of i depends on the value of RP. The cost of only certain binary instructions with one variable operand depends on the value of RP. For instance, the cost of the load instruction **ld** v, c is 2 or 3 bytes depending upon whether RP points to the bank in which v is stored or not, but the cost of the add instruction **add** v, c is independent of the value of RP.
- `IncrInstr` $\subseteq$ (`Instr`$_{abs}$ $\times$ `Var`) is the set of all increment instructions. We also define `Bin1orIncrInstr` = `Bin1Instr` $\cup$ `IncrInstr`.
- `DjnzInstr` $\subseteq$ (`Instr`$_{abs}$ $\times$ `Var`) is the set of all "decrement and jump if non-zero" instructions.

3.2 0–1 Variables

- The variables $r_{v,b}$ are defined such that for each $v \in$ `Var` and $b \in$ `Bank`, the ILP solver sets $r_{v,b}$ to 1 if it is desirable to store v in (any register in) bank b.
- The variables $\mathrm{RPVal}_{i,b}$ are defined such that for each $i \in$ `Instr`$_{abs}$ and $b \in$ `Bank`, the solver sets $\mathrm{RPVal}_{i,b}$ to 1 if it is desirable for the value of RP to be b whenever i is executed.
- The variables $\mathrm{DiffRP}_{i_1,i_2}$ are defined such that for each $(i_1, i_2) \in$ `EdgeOrCallPair`, $\mathrm{DiffRP}_{i_1,i_2}$ is 1 if $\exists b \in$ `Bank`. $\mathrm{RPVal}_{i_1,b} \neq \mathrm{RPVal}_{i_2,b}$.
- The variables SetRP_i are defined such that for each $i \in$ `Instr`$_{abs}$, the solver sets SetRP_i to 1 if it is desirable to introduce **srp** b immediately before i.
- The variables PuRP_i are defined such that for each $i \in$ `PuInstr`, the solver sets PuRP_i to 1 if it is desirable to introduce **push RP** immediately before i.
- The variables PoRP_i are defined such that for each $i \in$ `PoInstr`, the solver sets PoRP_i to 1 if it is desirable to introduce **pop RP** immediately before i.
- The variables $\mathrm{Bin2Cost}_i$ are defined such that for each $(i, v_1, v_2) \in$ `Bin2Instr`, $\mathrm{Bin2Cost}_i$ is 1 if $\exists b \in$ `Bank`. $\mathrm{RPVal}_{i,b} \neq r_{v_1,b} \vee \mathrm{RPVal}_{i,b} \neq r_{v_2,b}$.
- The variables $\mathrm{Bin1orIncrCost}_i$ are defined such that for each $(i, v) \in$ `Bin1orIncrInstr`, $\mathrm{Bin1orIncrCost}_i$ is 1 if $\exists b \in$ `Bank`. $\mathrm{RPVal}_{i,b} \neq r_{v,b}$.

3.3 Constraints

The general form of an integer linear constraint is

$$C_1V_1 + \ldots + C_nV_n \sim C$$

where $C_1, \ldots, C_n, C$ are integer constants, $V_1, \ldots, V_n$ are integer variables, and $\sim$ is one of $<$, $>$, $\leq$, $\geq$, and $=$. Our ILP formulation uses the following variants of the above form:

$$\begin{aligned} V &= 0 \\ V &= 1 \\ V &= V' \\ V &\leq V' \\ V_1 + \ldots + V_n &= 1 \\ V_1 + \ldots + V_n &\leq C \\ V &\leq V' + V'' \end{aligned}$$

where $V, V', V'', V_1, \ldots, V_n$ are variables ranging over $\{0, 1\}$. It is straightforward to show that it is NP-complete to decide solvability of a finite set of constraints of the above forms. We will use the abbreviation

$$|V' - V''| \leq V$$

to denote the two constraints

$$\begin{aligned} V' &\leq V + V'' \\ V'' &\leq V + V' \end{aligned}$$

3.3.1 Assigning Variables to Banks

Constraints (1–4) state that all predefined variables must be stored in their respective banks:

$$\forall v \in \texttt{PDV0}.\ r_{v,0} = 1 \quad (1)$$

$$\forall v \in \texttt{PDV0}.\ \forall b \in \texttt{Bank} - \{0\}.\ r_{v,b} = 0 \quad (2)$$

$$\forall v \in \texttt{PDV15}.\ r_{v,15} = 1 \quad (3)$$

$$\forall v \in \texttt{PDV15}.\ \forall b \in \texttt{Bank} - \{15\}.\ r_{v,b} = 0 \quad (4)$$

Constraint (5) states that a user-defined variable must be stored in exactly one bank:

$$\forall v \in \texttt{UDV}. \sum_{b \in \texttt{Bank}} r_{v,b} = 1 \tag{5}$$

The number of user-defined variables must not exceed the number of general-purpose registers. Since the 0^{th} bank has 12 such registers, the 15^{th} bank has none, and the rest have 16 each, we have:

$$\textstyle\sum_{v \in \texttt{UDV}} r_{v,0} \le 12 \tag{6}$$

$$\forall v \in \texttt{UDV}.\ r_{v,15} = 0 \tag{7}$$

$$\forall b \in \texttt{Bank} - \{0, 15\}.\ \textstyle\sum_{v \in \texttt{UDV}} r_{v,b} \le 16 \tag{8}$$

3.3.2 Introducing RP-manipulating instructions

Constraint (9) states that RP must be set to exactly one bank at every instruction:

$$\forall i \in \texttt{Instr}_{abs}. \sum_{b \in \texttt{Bank}} \mathrm{RPVal}_{i,b} = 1 \tag{9}$$

Constraint (10) states that if $\exists b \in \texttt{Bank}.\ \mathrm{RPVal}_{i_1,b} \ne \mathrm{RPVal}_{i_2,b}$, then $\mathrm{DiffRP}_{i_1,i_2} = 1$:

$$\forall (i_1, i_2) \in \texttt{EdgeOrCallPair}.\ \forall b \in \texttt{Bank}.$$
$$|\mathrm{RPVal}_{i_1,b} - \mathrm{RPVal}_{i_2,b}| \le \mathrm{DiffRP}_{i_1,i_2} \tag{10}$$

For $(i_1, i_2) \in \texttt{Edge}$ where $\mathrm{DiffRP}_{i_1,i_2} = 1$, we introduce an RP-modifying instruction in the following way, depending on the kind of edge:

1. For $(i_1, i_2) \in \texttt{MiscPair}$ where $\mathrm{DiffRP}_{i_1,i_2} = 1$, we introduce **srp** immediately before i_2:

   $$\forall (i_1, i_2) \in \texttt{MiscPair}.\ \mathrm{DiffRP}_{i_1,i_2} \le \mathrm{SetRP}_{i_2} \tag{11}$$

2. For $(i_1, i_2) \in \texttt{PRetEdge}$ where $\mathrm{DiffRP}_{i_1,i_2} = 1$, we introduce either **pop RP** immediately before i_1 or **srp** immediately before i_2 (or both):

   $$\forall (i_1, i_2) \in \texttt{PRetEdge}.$$
   $$\mathrm{DiffRP}_{i_1,i_2} \le \mathrm{PoRP}_{i_1} + \mathrm{SetRP}_{i_2} \tag{12}$$

3. For $(i_1, i_2) \in \texttt{IRetEdge}$ where $\mathrm{DiffRP}_{i_1,i_2} = 1$, we introduce **pop RP** immediately before i_1:

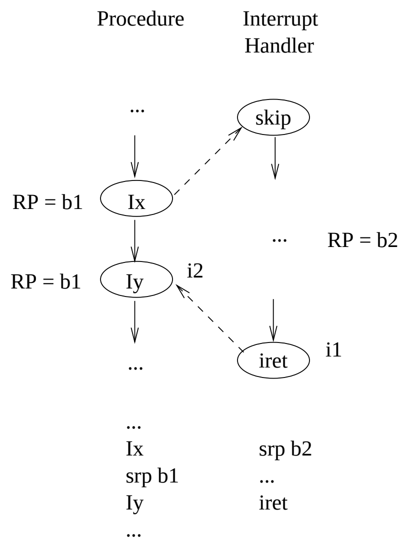
   $$\forall (i_1, i_2) \in \texttt{IRetEdge}.\ \mathrm{DiffRP}_{i_1,i_2} \le \mathrm{PoRP}_{i_1} \tag{13}$$

   In this case we do not have the option of introducing **srp** immediately before i_2; Figure 4 illustrates a scenario in which doing so results in incorrect behavior of the target program.

For $(i_1, i_4) \in \texttt{CallPair}$, the motivation behind computing $\mathrm{DiffRP}_{i_1,i_4}$ is as follows. There is an additional constraint on introducing **pop RP** in item (2) above, namely, for $(i_3, i_4) \in \texttt{PretEdge}$ such that $(i_1, i_2, i_3, i_4) \in \texttt{ProcQuad}$, the value of RP at the **call** instruction i_1 must be equal to the value of RP at instruction i_4 following it, *i.e.*, $\mathrm{DiffRP}_{i_1,i_4} = 0$.

$$\forall (i_1, i_2, i_3, i_4) \in \texttt{ProcQuad}.\ \mathrm{PoRP}_{i_3} + \mathrm{DiffRP}_{i_1,i_4} \le 1 \tag{14}$$

Figure 5 elucidates why the above constraint is needed on introducing **pop RP** immediately before i_3 for $(i_3, i_4) \in \texttt{PRetEdge}$ but not for $(i_3, i_4) \in \texttt{IRetEdge}$.



RP will be b2 (instead of b1) at Iy if the handler is invoked immediately after the execution of srp b1

Figure 4: Scenario

Constraint (15) states that **srp** cannot be introduced immediately before i_{root}.

$$\mathrm{SetRP}_{i_{root}} = 0 \tag{15}$$

Constraint (16) states that whenever we introduce **pop RP**, we must also introduce the matching **push RP**.

$$\forall (i_1, i_2) \in (\texttt{ProcPair} \cup \texttt{IntrPair}).\ \mathrm{PuRP}_{i_1} = \mathrm{PoRP}_{i_2} \tag{16}$$

3.3.3 Measuring Code Size

Any instruction in `Bin2Instr` occupies 2 or 3 bytes depending upon whether both operands are stored in working registers or not, respectively. The following constraints characterize the space cost of such instructions:

$$\forall (i, v_1, v_2) \in \texttt{Bin2Instr}.\ \forall b \in \texttt{Bank}.$$
$$|\mathrm{r}_{v_1,b} - \mathrm{RPVal}_{i,b}| \le \mathrm{Bin2Cost}_i \tag{17}$$

$$\forall (i, v_1, v_2) \in \texttt{Bin2Instr}.\ \forall b \in \texttt{Bank}.$$
$$|\mathrm{r}_{v_2,b} - \mathrm{RPVal}_{i,b}| \le \mathrm{Bin2Cost}_i \tag{18}$$

Thus, for any $(i, v_1, v_2) \in \texttt{Bin2Instr}$, v_1 and v_2 will be in working registers when i is executed if $\mathrm{Bin2Cost}_i = 0$.

Any instruction in `Bin1Instr` (`IncrInstr`) occupies 2 (1) or 3 (2) bytes depending upon whether the operand is stored in a working register or not, respectively. The following constraint characterizes the space cost of such instructions:

$$\forall (i, v_1, v_2) \in \texttt{Bin1OrIncrInstr}.\ \forall b \in \texttt{Bank}.$$
$$|\mathrm{r}_{v,b} - \mathrm{RPVal}_{i,b}| \le \mathrm{Bin1OrIncrCost}_i \tag{19}$$

Thus, for any $(i, v) \in \texttt{Bin1OrIncrInstr}$, v will be in a working register when i is executed if $\mathrm{Bin1OrIncrCost}_i = 0$.

The operand of any unary instruction in Z86E30's instruction set can be stored in a working or non-working register, with the exception of the **djnz** v, dst instruction: v *must* be stored in a working register. (Program memory addresses like dst are not treated as operands. So, **djnz** qualifies as a unary instruction.) This condition is enforced as flows:

$$\forall(i,v) \in \texttt{DjnzInstr}.\ \forall b \in \texttt{Bank}.\ r_{v,b} = \text{RPVal}_{i,b} \quad (20)$$

3.4 Objective Function

The objective of our ILP is to minimize the instruction space cost of the target program. The space cost of each instruction in `Bin2Instr` or `Bin1OrIncrInstr` depends upon whether its operands are stored in working registers or not. Also, the space cost of each of **srp**, **push RP**, and **pop RP** is 2 bytes. Thus, the objective function is:

$$\sum_{(i,v_1,v_2)\in\texttt{Bin2Instr}} \text{Bin2Cost}_i + \sum_{(i,v)\in\texttt{Bin1OrIncrInstr}} \text{Bin1OrIncrCost}_i +$$

$$\sum_{i\in\texttt{Instr}_{abs}} 2\ \text{SetRP}_i + \sum_{i\in\texttt{PuInstr}} 2\ \text{PuRP}_i + \sum_{i\in\texttt{PoInstr}} 2\ \text{PoRP}_i$$

We could dispense with the variable PuRP in our ILP formulation by eliminating constraint (16) and replacing PuRP in the objective function by PoRP. However, our experiments suggest that doing so increases ILP solution time significantly.

4. CODE GENERATION

Given a solution to the ILP generated from a ZIL program, we generate code for the Z86E30 as follows:

- If $v \in \texttt{UDV}$ and $r_{v,b} = 1$, then we store v in (any register in) bank b. Constraint (5) ensures that there is a unique such b.
- If $(i, v_1, v_2) \in \texttt{Bin2Instr}$, then the registers allocated to both v_1 and v_2 are addressed using the 4-bit or 8-bit addressing mode, depending upon whether Bin2Cost_i is 0 or 1, respectively.
- If $(i, v) \in \texttt{Bin1OrIncrInstr}$, then v is addressed using the 4-bit or 8-bit addressing mode, depending upon whether Bin1OrIncrCost_i is 0 or 1, respectively.
- If $(i, v) \in \texttt{DjnzInstr}$, then the register allocated to v is addressed using the 4-bit addressing mode.
- If $i \in \texttt{Instr} - \texttt{Instr}_{abs}$ and $v \in \texttt{Var}$ is an operand of i, then the register allocated to v is addressed using the 8-bit addressing mode.
- If $i \in \texttt{Instr}_{abs}$, $\text{SetRP}_i = 1$, and b is the bank such that $\text{RPVal}_{i,b} = 1$, then we introduce **srp** b immediately before i. Constraint (9) ensures that there is a unique such b.
- If $i_1 \in \texttt{PuInstr}$, $i_2 \in \texttt{PoInstr}$, $\text{PuRP}_{i_1} = 1$, and $\text{PoRP}_{i_2} = 1$, then we introduce **push RP** and **pop RP** immediately before i_1 and i_2, respectively. Constraint (16) ensures that **push RP** and **pop RP** are always introduced in matching pairs.
- If $\text{RPVal}_{i_{root},b} = 1$, then we replace i_{root} by **srp** b. Constraint (15) ensures that there is no **srp** instruction before i_{root}.

5. FOUR APPROACHES TO CODE-SIZE-DIRECTED COMPILATION

We have measured the performance of four techniques for register allocation and code-generation as follows.

5.1 SetRP + Full PuPoRP

This is the technique presented in Sections 2–4. It is the most liberal in terms of the kind of RP-manipulating instructions allowed and the locations in which they can be introduced.

5.2 SetRP + PuPoRP

This technique allows introducing **push/pop RP** only for interrupt handlers. CFG abstraction is aggressive: we define $\texttt{Instr}_{abs}$ as the subset of `Instr` such that each instruction in the former satisfies at least one of the two conditions:

- The instruction is i_{root}, an **iret**, a **skip** at the entry point of an interrupt handler, or **djnz**.
- The space cost of the instruction depends upon the value of RP.

5.3 SetRP

This technique does not allow introducing **push/pop RP** at all. CFG abstraction is even more aggressive: we define $\texttt{Instr}_{abs}$ as the subset of `Instr` such that each instruction in the former satisfies at least one of the two conditions:

- The instruction is i_{root} or **djnz**.
- The space cost of the instruction depends upon the value of RP.

Moreover, **srp** *cannot* be introduced immediately before an instruction in $\texttt{Instr}_{abs}$ that belongs to an interrupt handler. This is because interrupt handlers, especially in large programs, can be invoked immediately before executing different instructions that expect different values of RP during their respective executions. If RP is allowed to be modified within such a handler, it is not possible to restore its original value using **srp** before returning from the handler.

5.4 Cheap

This technique causes a single **srp** to be introduced at the start of the program; after that RP is not changed at all. It is based on an ILP formulation that declares the sets `Var`, `PDV0`, `PDV15`, `Bin2Instr`, `Bin1OrIncrInstr`, and `DjnzInstr` as before; two kinds of variables: Bin2Cost, as before, and the variable InCurrBank for each $v \in \texttt{Var}$ such that InCurrBank_v is set to 1 if it is desirable to store v in the bank to which RP points; and the following constraints:

$$\sum_{v\in\texttt{Var}} \text{InCurrBank}_v \leq 16$$

$$\forall v_1 \in \texttt{PDV0}.\quad \forall v_2 \in \texttt{PDV0}.\quad \text{InCurrBank}_{v_1} = \text{InCurrBank}_{v_2}$$

$$\forall v_1 \in \texttt{PDV15}.\ \forall v_2 \in \texttt{PDV15}.\ \text{InCurrBank}_{v_1} = \text{InCurrBank}_{v_2}$$

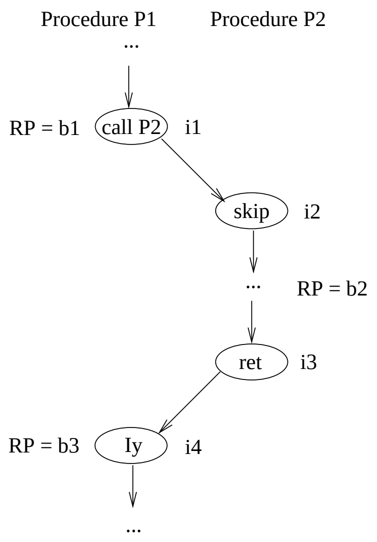
$$\forall v_1 \in \texttt{PDV0}.\quad \forall v_2 \in \texttt{PDV15}.$$

$$\text{InCurrBank}_{v_1} + \text{InCurrBank}_{v_2} \leq 1$$

$$\forall(i,v_1,v_2) \in \texttt{Bin2Instr}.\ \text{Bin2Cost}_i + \text{InCurrBank}_{v_1} \geq 1$$

$$\forall(i,v_1,v_2) \in \texttt{Bin2Instr}.\ \text{Bin2Cost}_i + \text{InCurrBank}_{v_2} \geq 1$$

$$\forall(i,v) \in \texttt{DjnzInstr}.\ \text{InCurrBank}_v = 1$$



Since RP at i3 differs from RP at i4, constraint (12) allows pop RP or srp b3 to be introduced immediately before i3 and i4, respectively.

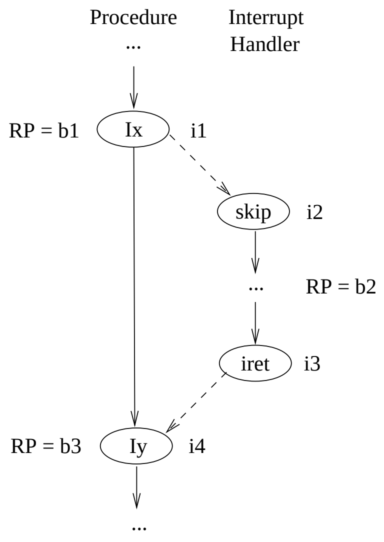
In the absence of constraint (14), the former choice results in the following incorrect target code if RP at i1 differs from RP at i4:

```
...            push RP
call P2        srp b2
Iy             ...
...            pop RP
               ret
```

The push RP is introduced by constraint (16). Note that RP is b1 (instead of b3) at Iy in the target code, which is incorrect.

In the presence of constraint (14), there is no choice but to introduce srp b3 immediately before i4, resulting in the following (correct) target code:

```
...            srp b2
call P2        ...
srp b3         ret
Iy
```



Since RP at i3 differs from RP at i4, constraint (13) allows pop RP to be introduced immediately before i3.

This results in the following (correct) target code even if RP at i1 differs from RP at i4:

```
...        push RP
Ix         srp b2
srp b3     ...
Iy         pop RP
...        iret
```

This is because constraint (11) accounts for the change in RP from i1 to i4 by introducing srp b3 immediately before i4.

Figure 5: Scenario

The objective function is to minimize:

$$\sum_{(i,v_1,v_2)\in\texttt{Bin2Instr}} \text{Bin2Cost}_i \quad - \sum_{(i,v)\in\texttt{Bin1OrIncrInstr}} \text{InCurrBank}_v$$

6. EXPERIMENTAL RESULTS

6.1 Benchmark Characteristics

For our experiments, we have used the example program in Appendix A (`example.zil`) and two proprietary microcontroller programs, called `serial.zil` and `cturk.zil`, provided by Greenhill Manufacturing, Ltd. [2]. Greenhill has over a decade of experience producing environmental control systems for agricultural needs. Some characteristics of these programs are presented in Figure 6. For `example.zil`, the number of nodes in CFG$_{abs}$ is more than that in CFG because, during model extraction, 4 **skip** instructions are introduced in the first step (skip insertion), and only one instruction is eliminated in the fourth step (abstraction).

6.2 Measurements

Figure 7 presents our experimental results. The solution time for the ILP-based techniques was measured on a 1.1 GHz Intel Pentium IV machine with 512 MB RAM (though CPLEX was limited to use at most 128 MB).

The results for `cturk.zil` were obtained by reducing the number of banks to 6, *i.e.*, by defining $\texttt{Bank} = \{0..4, 15\}$, in order to reduce the number of equivalent solutions explored by CPLEX. Since `cturk.zil` contains 55 user-defined variables and the six chosen banks have 60 general-purpose registers, increasing the number of banks would not have reduced the size of the target code.

In Figure 7, in the row for the number of instructions addressing only working registers, the "upper bound" indicates the number of instructions whose space cost depends on the value of RP.

Number of:	`example`	`serial`	`cturk`
Lines of ZIL (nodes in CFG)	14	181	850
Lines of ZIL after abstraction (nodes in CFG$_{abs}$)	17	53	304
Edges in CFG$_{abs}$	42	193	1287
Instructions in `Bin2Instr`	1	9	147
Instructions in `Bin1Instr`	2	41	126
Instructions in `IncrInstr`	1	2	20
Instructions in `DjnzInstr`	2	0	10
User-defined variables	5	9	55
Procedures (excluding MAIN)	2	6	37
Interrupt handlers	1	2	2

Figure 6: Benchmark characteristics

Number of:	technique	`example`	`serial`	`cturk`
Integer variables	Cheap	23	33	187
	SetRP	401	1035	2343
	SetRP + PuPoRP	449	1275	2885
	SetRP + Full PuPoRP	617	2009	6909
Constraints	Cheap	225	239	525
	SetRP	656	3132	8900
	SetRP + PuPoRP	1052	6369	21426
	SetRP + Full PuPoRP	1722	9953	35961
Seconds to solve the ILP	Cheap	0.00	0.01	0.10
	SetRP	0.04	0.27	856
	SetRP + PuPoRP	0.07	0.92	2478
	SetRP + Full PuPoRP	0.14	3.39	59351
srp introduced	Cheap	1	1	1
	SetRP	1	2	19
	SetRP + PuPoRP	1	2	21
	SetRP + Full PuPoRP	1	2	18
push/pop RP introduced	SetRP + PuPoRP	0	0	2
	SetRP + Full PuPoRP	0	0	2
instructions addressing only working registers	Cheap	4	29	50
	SetRP	3	30	131
	SetRP + PuPoRP	4	35	150
	SetRP + Full PuPoRP	4	35	150
	upper bound	4	52	293

Figure 7: Measurements

6.3 Assessment

As expected, the ILP solution time increases significantly and the space savings increases monotonically from SetRP through SetRP + PuPoRP to SetRP + Full PuPoRP.

The space savings in SetRP + Full PuPoRP is the same as that in SetRP + PuPoRP for all the benchmark programs because none of the procedures in our benchmark programs is both large and called from several sites having different RP values so as to offset the high overhead associated with introducing **push/pop RP**. Moreover, the former takes significantly more time to solve than the latter. Thus, SetRP + PuPoRP offers the best trade-off between solution time and space savings.

Figure 8 compares the sizes of the Z86 programs generated using SetRP + PuPoRP against the sizes of the handwritten Z86 programs. For `serial.zil`, the code generated using our technique is superior to the handwritten code. However, for `cturk.zil`, the code generated using our technique is bigger even though our technique performs superior register allocation. This is because the handwritten `cturk` program uses programming tricks such as jumping from one routine to instructions within another routine. While that can lead to more compact code, it is forbidden in ZIL.

7. CONCLUSION

We have explored four ILP-based approaches to code-size-directed compilation. The SetRP + PuPoRP approach provides a good trade-off between solution time and space savings. Our code-size-directed compiler generates code that is as compact as carefully crafted code. Since ILP is NP-complete, generating optimally-sized target code for large programs can be prohibitively time consuming. However, it is possible to simply state a desired upper bound on target code size and require the generated code to be *small enough*; such code may have a chance of being generated in reasonable time.

Size (in bytes) of:	serial	cturk
handwritten Z86 code	415	1789
Z86 code generated using SetRP + PuPoRP	382	1811

Figure 8: Code-size comparison

8. ACKNOWLEDGMENTS

We thank James Rose for rewriting the Z86 version of `cturk` in ZIL and Matthew Wallace for writing an interpreter for ZIL. We thank Dennis Brylow for help in numerous situations. Palsberg is supported by a National Science Foundation Information Technology Research Award number 0112628.

9. REFERENCES

[1] CPLEX mixed integer optimizer. www.ilog.com/products/cplex/product/mip.cfm.

[2] Greenhill Manufacturing. www.greenhillmfg.com.

[3] Zilog, Inc. www.zilog.com.

[4] A. Appel and L. George. Optimal spilling for CISC machines with few registers. In C. Norris and J. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, ACM SIGPLAN Notices, pages 243–253. ACM Press, June 2001.

[5] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, pages 47–56, Toronto, May 2001.

[6] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* The Scientific Press, 1993. www.ampl.com.

[7] D. Goodwin and K. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software—Practice and Experience*, 26(8):929–965, Aug. 1996.

[8] T. Kong and K. D. Wilken. Precise register allocation for irregular register architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307. IEEE Computer Society, Nov. 30–Dec. 2 1998.

[9] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proceedings of IEEE International Conference on Computer Aided Design*, 1996.

[10] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[11] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. *Lecture Notes in Computer Science*, 1575:137–152, 1999.

[12] R. B. O. Avissar and D. Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the ACM 2nd Int'l Conf. on Compilers, Architectures, and Synthesis for Embedded Systems CASES*, Nov. 2001.

[13] J. Park, J. Lee, and S. Moon. Register allocation for banked register file. In C. Norris and J. Fenwick, editors, *Proceedings of the Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES-01)*, volume 36, 6 of *ACM SIGPLAN Notices*, pages 39–47. ACM Press, June 22–23 2001.

[14] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 128–138, May 1–4, 1999.

[15] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 1–11, 1996.

[16] J. Sjdin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of CASES 2001*, pages 15–23, 2001.

[17] A. Stoutchinin. An integer linear programming model of software pipelining for the MIPS R8000 processor. *Lecture Notes in Computer Science*, 1277:121–135, 1997.

[18] A. Sudarsanam and S. Malik. Simultaneous reference allocation in code generation for dual data memory bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems.*, 5(2):242–264, Jan. 2000.

APPENDIX A: ZIL

The grammar for ZIL is shown in Figure 9. In ZIL, variables can be declared global, local to the main program, or local to any procedure or interrupt handler. There are 16 predefined global variables: P0, P1, P2, and P3, which are allotted the 4 special-purpose registers in the 0^{th} bank, and FLAGS, T0, T1, P01M, P2M, P3M, TMR, PRE0, PRE1, IMR, IPR, and IRQ, which are allotted 12 of the 16 special-purpose registers in the 15^{th} bank. Each ZIL routine (procedure or interrupt handler) has a single entry point and a single exit point. Each procedure has a single **ret** instruction and each interrupt handler has a single **iret** instruction. Recursion, both direct and indirect, and jumps from one routine to an instruction within another routine are disallowed. The ZIL instructions are, essentially, Z86E30 instructions except that they operate on variables instead of registers. Figure 10 shows an example ZIL program.

```
             Goal ::= ( GlobalDef )* "MAIN" MainBlock "PROCEDURES" ( ProcDef )*
                      "HANDLERS" ( HandlerDef )*
        GlobalDef ::= ConstDef | VarDef
         ConstDef ::= "static" "final" Type Id "=" Literal
           VarDef ::= Type Variable
             Type ::= "int" | "string" | "proc_label" | "jump_label"
          ProcDef ::= Label "(" ")" ProcedureBlock
       HandlerDef ::= Label "(" ")" HandlerBlock
        MainBlock ::= "{" ( VarDef )* ( Stmt )* "}"
   ProcedureBlock ::= "{" ( VarDef )* ( Stmt )* ( Label ":" )? "RET"  "}"
     HandlerBlock ::= "{" ( VarDef )* ( Stmt )* ( Label ":" )? "IRET" "}"
             Stmt ::= ( Label ":" )? Instruction
      Instruction ::= ArithLogic1aryOPC Variable
                   |  ArithLogic2aryOPC Variable "," Expr
                   |  CPUControl1aryOPC Expr
                   |  CPUControl0aryOPC
                   |  "LD" Variable "," LDExpr
                   |  "DJNZ" Variable "," Label
                   |  "JP" ( Condition "," )? LabelExpr
                   |  "CALL" LabelExpr
                   |  "preserveIMR" "{" ( Stmt )* ( Label ":" )? "}"
           LDExpr ::= "@" Id | Expr | "LABEL" Label
             Expr ::= "!" Expr | "(" Expr "&" Expr ")" | "(" Expr "|" Expr ")" |  Prim
             Prim ::= Id | IntLiteral
        LabelExpr ::= Label | "@" Variable
         Variable ::= Id
            Label ::= Id
ArithLogic1aryOPC ::= "CLR" | "COM" | "DA"  | "DEC" | "INC" | "POP"  | "PUSH"
                   |  "RL"  | "RLC" | "RR"  | "RRC" | "SRA" | "SWAP"
ArithLogic2aryOPC ::= "ADC" | "ADD" | "AND" | "CP"  | "OR"  | "SBC"  | "SUB" | "TCM" | "TM" | "XOR"
CPUControl0aryOPC ::= "CLRIMR" | "CLRIRQ" | "EI"    | "DI"  | "HALT" | "NOP"
                   |  "RCF"    | "SCF"    | "STOP"  | "WDH" | "WDT"
CPUControl1aryOPC ::= "ANDIMR" | "ANDIRQ" | "LDIMR" | "LDIPR" | "LDIRQ"
                   |  "ORIMR"  | "ORIRQ"  | "TMIMR" | "TMIRQ"
        Condition ::= "F"  | "C"  | "NC" | "Z"  | "NZ" | "PL"  | "MI"  | "OV"  | "NOV" | "EQ"
                   |  "NE" | "GE" | "GT" | "LE" | "LT" | "UGE" | "ULE" | "ULT" | "UGT" | "GLE"
          Literal ::= IntLiteral | StrLiteral
       IntLiteral ::= <HEX_H> | <BIN_B> | <DEC_D>
       StrLiteral ::= <STRING_CONSTANT>
               Id ::= <IDENTIFIER>
```

Figure 9: The ZIL grammar

```
int intrs

MAIN
{
     int   x
     int   y
START:
     cp    x, y
     jp    eq, L0
     call  T4
     jp    L1
L0:  call  T8
L1:  jp    START

}
```

```
PROCEDURES

T4()
{
     int   u
     ld    u, 04h
L2:  djnz  u, L2
     ret
}

T8()
{
     int   v
     ld    v, 08h
L3:  djnz  v, L3
     ret
}
```

```
HANDLERS

INTR()
{
     inc   intrs
     iret
}
```

Figure 10: example.zil