Compiling with Code-Size Constraints

MAYUR NAIK and JENS PALSBERG
Purdue University

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 that is difficult to maintain and reuse. In this paper, we present a step towards code-size-aware compilation. 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. The resulting compiler, when applied to six commercial microcontroller programs, generates code nearly as compact as carefully crafted code.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Banked architecture, integer linear programming, register allocation, space optimization

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.

Authors' address: Department of Computer Science, Purdue University, West Lafayette, IN 47907; email: {mnaik,palsberg}@cs.purdue.edu.

Authors	Architecture	Good data layout increases the opportunities for using:
[Liao et al. 1996] [Leupers and Marwedel 1996] [Rao and Pande 1999]	contemporary digital signal processor	autoincrement/autodecrement addressing modes
[Sudarsanam and Malik 2000]	two memory units	parallel data access modes
[Sjödin and von Platen 2001]	multiple address spaces	pointer addressing modes
[Park et al. 2001]	two register banks	RP-relative addressing
this paper	multiple register banks	RP-relative addressing

Fig. 1. Good data layout can significantly reduce code size.

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 that 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 et al. [1996] 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-aware compiler that generates code which is nearly 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 et al. [2001] 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. 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

```
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.

1.2 The Problem: Compiling for the Z86E30

We address the problem of code-size-aware compilation for Zilog's Z86E30 microcontroller [Zilog 1999]. The Z8 family of microcontrollers continues to be widely used, for example, in the pacemakers from Guidant Corporation (the second largest producer of pacemakers in the United States). It is straightforward to adapt our technique to other banked register architectures, for instance, the Intel 8051 microcontroller.

We focus on compiling ZIL to Z86 assembly. ZIL is a high-level language we have designed that resembles C in some respects. ZIL supports interrupt handling and provides low-level instructions for managing interrupts. The grammar of ZIL together with some notes on the semantics is presented in the appendix, and an example ZIL program is discussed in Section 2.

A key problem to a compiler is the Z86E30's lack of a data/stack memory: all variables declared in a ZIL program must be stored in registers. Moreover, it is a major challenge to distribute the variables among the various banks in a manner that reduces the overall space cost. As a result, whole-program register allocation must be performed, as opposed to per-procedure register allocation.

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 four registers in the 0th bank and the 16 registers in the 15th bank, are special purpose. The special-purpose registers are accessible to the ZIL programmer by way of predefined variables.

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 decrement instruction **dec** *v* or the add instruction **add** *v*, *c* (*c* is a constant) is independent of the value of RP. But the cost of the increment instruction **inc** *v* or the 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. We shall designate such instructions *RP sensitive*. About 30% of the instructions in our benchmark programs are RP sensitive. Thus, good register allocation is the key to compacting the target code generated for a ZIL program.

1.3 Our Results

We have designed and implemented a code-size-aware compiler from ZIL to Z86 assembly language. Our benchmark suite consists of six proprietary microcontroller programs, provided by Greenhill Manufacturing that were carefully handwritten in Z86 assembly. We converted the assembly programs to ZIL

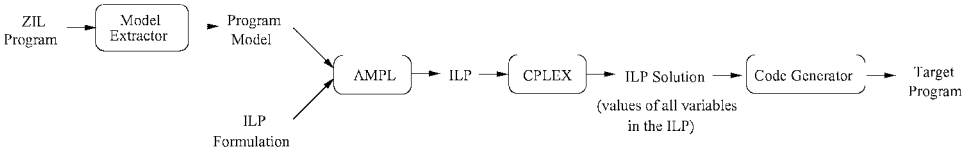

Fig. 2. Overview.

programs using an automatic reverse engineering tool [Palsberg and Wallace 2002], and then compiled the ZIL programs back to Z86 assembly programs using our code-size-aware compiler. This enables a direct comparison of the size of the generated code to that of the original handwritten code.

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 six microcontroller programs, generates code that is nearly as compact as the original handwritten code.

Our fully automated approach is illustrated in Figure 2. For a given ZIL program, we first perform model extraction to derive a control-flow graph. Next, we use the AMPL tool [Fourer et al. 1993] 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 the CPLEX ILP solver [ILOG 2001]. Once a solution has been obtained, a code generator produces a target Z86 assembly program.

We have experimented with three ILP formulations:

—Inexpensive allows only one **srp** right at the start of the program.
—Selective restricts **srp** to the entry and exit points of procedures and interrupt handlers and **push** (**pop**) **RP** to the entry (exit) points of interrupt handlers.
—Exhaustive allows **srp** to be introduced before any instruction in the model extracted from the ZIL program and restricts **push** (**pop**) **RP** to the entry (exit) points of interrupt handlers. An ILP formulation permitting **push** (**pop**) **RP** at the entry (exit) points of interrupt handlers *and* procedures is presented in Naik and Palsberg [2002]. We do not discuss it here because it is worthwhile only in the presence of procedures that are both large and called from several call sites. None of our benchmark programs have such procedures.

Our experiments show that Selective offers a good trade-off between ILP solving time and code-space savings.

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) for the following reasons:

—Different optimizations such as register allocation and instruction scheduling can be combined and solved within the same ILP framework, thereby avoiding the “phase-ordering problem” [Motwani et al. 1995; Kremer 1997].

—ILP guarantees optimality of the solution as opposed to heuristic methods [Ruttenberg et al. 1996].

—Even though ILP is NP-complete, significant advances in the integer and combinatorial optimization field (namely, branch-and-bound heuristics and cutting-plane technology), in conjunction with improvements in computation speed, have enabled its effective application to compiler optimizations [Kremer 1997].

Goodwin and Wilken [1996] pioneered the use of ILP for register allocation. Their approach is applicable only to processors with uniform register architectures. Kong and Wilken [1998] present an ILP framework for irregular register architectures (architectures that place restrictions on register usage). Appel and George [2001] 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 [1997] and Ruttenberg et al. [1996] present an ILP framework for integrated register allocation and software pipelining in the MIPS R8000 microprocessor. Liberatore et al. [1999] perform local register allocation (LRA) using ILP. Their experiments indicate 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 suboptimal.

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 that of Sjödin and Platen [2001] that models multiple address spaces of a certain processor using ILP, and that of Avissar et al. [2001] that uses ILP to optimally allocate global and stack data among different heterogeneous memory modules.

1.5 Our Approach

Figure 3 elucidates the simplest of our three ILP formulations, namely, Inexpensive, that allows only one **srp** *b* right at the start of the program. The formulation consists of set declarations, 0-1 variable declarations, an objective function, and linear constraints; the former two components are elided in the figure but explained below.

`Var` denotes the set of all variables in the ZIL program; `PDV0` $\subset$ `Var` and `PDV15` $\subset$ `Var` denote the sets of predefined variables for which special-purpose registers have been allotted in the 0th and 15th banks, respectively; `Bin2Instr` and `Bin1Instr` are the sets of RP-sensitive instructions in the program with two and one variable operands, respectively; and `DjnzInstr` is the set of the unary "Decrement and Jump if Non-Zero" (**djnz**) instructions. Recall that an RP-sensitive instruction occupies one fewer byte if its variable operand(s) are allotted registers in the bank to which RP points, namely, bank *b*. For each $(i, v_1, v_2) \in$ `Bin2Instr`, the 0-1 variable Bin2Cost_i is set to 0 or 1 by the ILP solver depending upon whether v_1 and v_2 are allotted registers in bank *b* or not, respectively. Likewise, for each $(i, v) \in$ `Bin1Instr`, the 0-1 variable InCurrBank_i

(*objective function*) Minimize:

$$\sum_{(i,v_1,v_2)\in \texttt{Bin2Instr}} \text{Bin2Cost}_i \quad - \sum_{(i,v)\in \texttt{Bin1Instr}} \text{InCurrBank}_v$$

(*linear constraints*) Subject to:

$$\sum\nolimits_{v\in \texttt{Var}} \text{InCurrBank}_v \leq 16 \tag{1}$$

$$\forall v_1 \in \texttt{PDV0}.\ \forall v_2 \in \texttt{PDV15}.\ \text{InCurrBank}_{v_1} + \text{InCurrBank}_{v_2} = 1 \tag{2}$$

$$\forall v_1 \in \texttt{PDV0}.\ \forall v_2 \in \texttt{PDV0}.\ \text{InCurrBank}_{v_1} = \text{InCurrBank}_{v_2} \tag{3}$$

$$\forall v_1 \in \texttt{PDV15}.\ \forall v_2 \in \texttt{PDV15}.\ \text{InCurrBank}_{v_1} = \text{InCurrBank}_{v_2} \tag{4}$$

$$\forall (i, v) \in \texttt{DjnzInstr}.\ \text{InCurrBank}_v = 1 \tag{5}$$

$$\forall (i, v_1, v_2) \in \texttt{Bin2Instr}.\ \text{Bin2Cost}_i + \text{InCurrBank}_{v_1} \geq 1 \tag{6}$$

$$\forall (i, v_1, v_2) \in \texttt{Bin2Instr}.\ \text{Bin2Cost}_i + \text{InCurrBank}_{v_2} \geq 1 \tag{7}$$

Fig. 3. The Inexpensive ILP formulation.

is set to 0 or 1 depending upon whether v is allotted a register in bank b or not, respectively.

The objective function seeks to minimize the number of bytes occupied by RP-sensitive instructions, one byte per instruction.

Constraint (1) states that at most 16 variables can be stored in bank b. Constraint (2) states that variables in PDV0 and PDV15 cannot be in bank b simultaneously. Constraint (3) states that variables in PDV0 must simultaneously be either in bank b or in a bank other than b; constraint (4) states the same for variables in PDV15. Constraint (5) states that the operand of each **djnz** instruction *must* be stored in bank b. This is a requirement imposed by the Z86E30 architecture. Constraint (6) states that for each $(i, v_1, v_2) \in$ Bin2Instr, Bin2Cost_i is 1 if v_1 is stored in a bank other than b; constraint (7) states the same for v_2.

1.6 Organization of the Paper

In Section 2, we present an example that illustrates the definitions and techniques used later in the paper. In Section 3, we explain model extraction. In Section 4, we present the Exhaustive ILP formulation. In Section 5, we discuss code generation. In Section 6, we present the Selective ILP formulation and our experimental results. Finally, Section 7 offers our conclusions.

2. EXAMPLE

An example ZIL program, called `example.zil`, is shown in Figure 4. The `MAIN` part of the program is an infinite loop that calls procedure `T4` or `T8`, depending upon whether `x` is equal to `y` or not, respectively. `T4` and `T8` simulate delay timers. `INTR` is an interrupt handler that simply counts the number of interrupts by incrementing the global variable `intrs`. We assume that interrupt handling is enabled in both the `MAIN` part and in the body of each of `T4` and `T8`, but not in the body of the interrupt handler.

From `example.zil`, we build a control-flow graph. To make the later stages more efficient, we also do an abstraction of the graph that may eliminate nodes and edges which do not play a role in register allocation. For `example.zil`, the only instruction eliminated by the abstraction is the conditional jump

```
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
}
```

Fig. 4. `example.zil`.

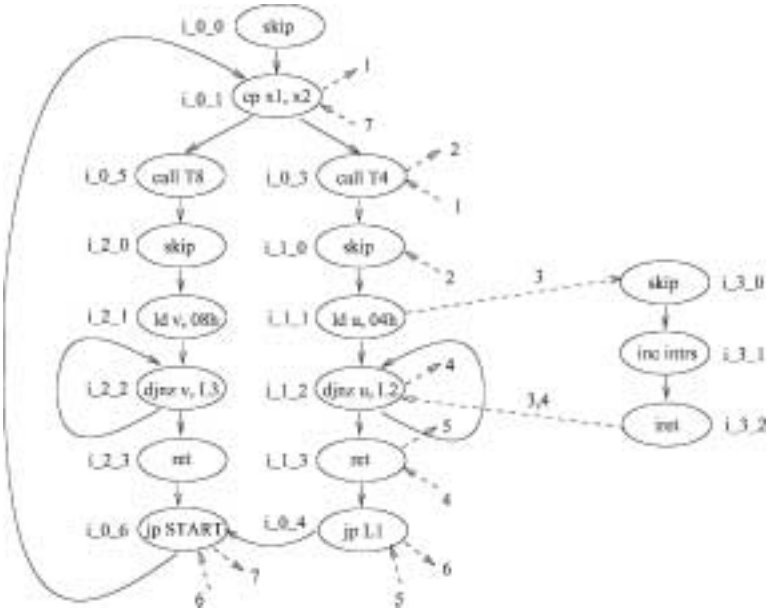

Fig. 5. Abstracted control-flow graph for `example.zil`.

instruction in `MAIN`. The abstracted control-flow graph is shown in Figure 5. Instructions are labeled `i_x_y`, where *x* is the number of the routine to which the instruction belongs and *y* is the number of the instruction within that routine. (A routine is a procedure or an interrupt handler.) Routines are numbered in the order in which they appear in the program; instructions in a routine are numbered in the order in which they appear in that routine. Dotted edges represent interrupt handler invocations and returns. To avoid cluttering in the illustration, those edges are not illustrated for `T8`; they are similar to those for `T4`.

```
MAIN                              PROCEDURES                    HANDLERS
{
    ; x, y are allotted registers  T4()                         INTR()
    ; 0, 1 in bank 1               {                            {
                                       ; u is allotted register     ; intrs is allotted register
START:                                 ; 0 in bank 2                ; 0 in bank 3
    srp 2
    cp  R10, R11  ; 1b wasted          ld   r0, 04h  ; 1b saved     inc R30  ; 1b wasted
    jp  eq, L0                     L2: djnz r0, L2                  iret
    call T4                            ret                      }
    jp  L1                         }
L0: call T8
L1: jp  START                      T8()
}                                  {
                                       ; v is allotted register
                                       ; 1 in bank 2

                                       ld   r1, 08h  ; 1b saved
                                   L3: djnz r1, L3
                                       ret
                                   }
```

Fig. 6. Target Z86 code for `example.zil`.

The abstracted control-flow graph is the program model, and it is represented in the form of set definitions. We show some of these definitions here:

$$\begin{aligned}
\texttt{Var} &:= \{\ \texttt{x, y, u, v, intrs}\ \} \\
\texttt{Bin2Instr} &:= \{\ \texttt{i_0_1}\ \} \\
\texttt{Bin1Instr} &:= \{\ \texttt{i_1_1, i_2_1, i_3_1}\ \} \\
\texttt{DjnzInstr} &:= \{\ \texttt{i_1_2, i_2_2}\ \}.
\end{aligned}$$

An ILP formulation is a template for ILPs. The ILP itself is generated from the program model and the ILP formulation. Since `example.zil` is small and uses only five variables, we will assume two registers per bank to make the problem of register allocation nontrivial.

From a solution to the ILP, we can do register allocation and code generation. Figure 6 shows a target Z86 assembly program based on a solution to the ILP generated from the program model of `example.zil` and the Inexpensive ILP formulation. A working register is denoted `r`k, where k is the register number in the current bank and a nonworking register is denoted `R`bk, where k is the register number in bank b. Both b and k are base-16 digits. Recall that Inexpensive allows only one **srp** b right at the start of the program ($b = 2$ in this example). Since the operand of every **djnz** instruction *must* be allotted a working register, variables u and v are allotted registers in bank 2.

3. MODEL EXTRACTION

Model extraction proceeds in the following four steps.

Skip insertion. First, we add an unlabeled **skip** instruction at the entry point of each routine; the one introduced at the entry point of `MAIN` is denoted

i_{root}. The extra **skip** instructions ensure that there are no jumps to the instruction at the entry point of a routine.

IMR estimation. Second, we conservatively estimate, for each program point, the interrupt handlers that are enabled, that is, the value of the interrupt mask register (IMR). We use a technique similar to the one described by Brylow et al. [2001].

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

Abstraction. Fourth, we define $\texttt{Instr}_{abs}$ as the subset of Instr such that each instruction in $\texttt{Instr}_{abs}$ satisfies at least one of the following two conditions:

—The instruction is a **ret**, **iret**, a **skip** at the entry point of a routine, or **djnz**.
—The instruction is RP-sensitive.

From CFG, we define an abstracted control-flow graph CFG_{abs} in which the 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 that does not contain any other node in $\texttt{Instr}_{abs}$. Thus, i_2 can be executed after i_1 but before any other instruction in $\texttt{Instr}_{abs}$ in a run of the original program.

4. THE EXHAUSTIVE ILP FORMULATION

4.1 Set Declarations

(1) $\texttt{Bank} = \{0, \ldots, 12, 15\}$ is the set of banks in the register file of the Z86E30; the 13th and 14th banks are reserved for the run-time stack.

(2) $\texttt{Edge} \subseteq (\texttt{Instr}_{abs} \times \texttt{Instr}_{abs})$ is the set of edges in CFG_{abs}.

(3) $\texttt{IretInstr} \subseteq \texttt{Instr}_{abs}$ is the set of **iret** instructions.

(4) $\texttt{IretEdge} = \{(i_1, i_2) \mid i_1 \in \texttt{IretInstr} \wedge (i_1, i_2) \in \texttt{Edge}\}$ and $\texttt{MiscEdge} = \texttt{Edge} - \texttt{IretEdge}$.

(5) 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.

(6) $\texttt{PDV0} \subseteq \texttt{Var}$ and $\texttt{PDV15} \subseteq \texttt{Var}$ are the sets of predefined variables for which special-purpose registers have been allotted in the 0th and 15th bank, respectively.

(7) $\texttt{Bin2Instr} \subseteq (\texttt{Instr}_{abs} \times \texttt{Var} \times \texttt{Var})$ is the set of triples (i, v_1, v_2) such that i is an RP-sensitive instruction with source and destination operands v_1 and v_2.

(8) $\texttt{Bin1Instr} \subseteq (\texttt{Instr}_{abs} \times \texttt{Var})$ is the set of pairs (i, v) such that i is an RP-sensitive instruction with source and destination operand v.

(9) $\texttt{DjnzInstr} \subseteq (\texttt{Instr}_{abs} \times \texttt{Var})$ is the set of all "Decrement and Jump if Non-Zero" instructions.

4.2 0–1 Variables

(1) The variables $r_{v,b}$ are defined such that for each $v \in \texttt{Var}$ and $b \in \texttt{Bank}$, the ILP solver sets $r_{v,b}$ to 1 if it is desirable to store v in (any register in) bank b.
(2) The variables $\text{RPVal}_{i,b}$ are defined such that for each $i \in \texttt{Instr}_{abs}$ and $b \in \texttt{Bank}$, the solver sets $\text{RPVal}_{i,b}$ to 1 if it is desirable that the value of RP is b whenever i is executed.
(3) The variables SetRP_i are defined such that for each $i \in \texttt{Instr}_{abs}$, the solver sets SetRP_i to 1 if it is desirable to introduce the instruction **srp** b (set RP to bank b) immediately before i.
(4) The variables PopRP_i are defined such that for each $i \in \texttt{IretInstr}$, the solver sets PopRP_i to 1 if it is desirable to introduce the instruction **pop RP** immediately before i (and the instruction **push RP** immediately immediately before the first instruction of the interrupt handler to which i belongs).
(5) The variables Bin2Cost_i are defined such that for each $(i, v_1, v_2) \in \texttt{Bin2Instr}$, Bin2Cost_i is 1 if $\exists b \in \texttt{Bank}.\ \text{RPVal}_{i,b} \neq r_{v_1,b} \vee \text{RPVal}_{i,b} \neq r_{v_2,b}$.
(6) The variables Bin1Cost_i are defined such that for each $(i, v) \in \texttt{Bin1Instr}$, Bin1Cost_i is 1 if $\exists b \in \texttt{Bank}.\ \text{RPVal}_{i,b} \neq r_{v,b}$.

4.3 Constraints

The general form of an ILP constraint is $C_1V_1 + \cdots + C_nV_n \sim C$, where $C_1, \ldots, C_n, C$ are integers, $V_1, \ldots, V_n$ are integer variables, and $\sim$ is one of $<$, $>$, $\leq$, $\geq$, and $=$. Our ILP formulation uses the following six variants of the above form:

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

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

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

to denote the two constraints

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

4.3.1 *Assigning Variables to Banks.* Constraint (8) states that a variable must be stored in exactly one bank:

$$\forall v \in \texttt{Var}.\ \sum_{b \in \texttt{Bank}} r_{v,b} = 1. \tag{8}$$

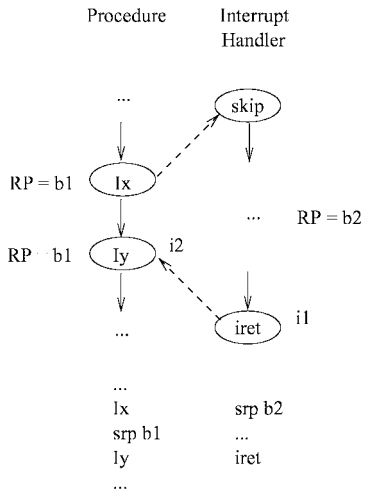

Fig. 7. Scenario.

Constraints (9) and (10) state that all predefined variables must be stored in their respective banks:

$$\forall v \in \texttt{PDV0}.r_{v,0} = 1 \tag{9}$$

$$\forall v \in \texttt{PDV15}.r_{v,15} = 1. \tag{10}$$

The total number of variables must not exceed the total number of registers. Since each bank has 16 registers, we have

$$\forall b \in \texttt{Bank}. \sum_{v \in \texttt{Var}} r_{v,b} \leq 16. \tag{11}$$

4.3.2 *Introducing RP-Manipulating Instructions.* Constraint (12) states that RP must be set to exactly one bank at every instruction:

$$\forall i \in \texttt{Instr}_{abs}. \sum_{b \in \texttt{Bank}} \text{RPVal}_{i,b} = 1. \tag{12}$$

Constraint (13) states that for each $(i_1, i_2) \in \texttt{MiscEdge}$, if $\exists b \in \texttt{Bank}.\text{RPVal}_{i_1,b} \neq \text{RPVal}_{i_2,b}$, then $\text{SetRP}_{i_2} = 1$:

$$\forall (i_1, i_2) \in \texttt{MiscEdge}.\ \forall b \in \texttt{Bank}.\ |\text{RPVal}_{i_1,b} - \text{RPVal}_{i_2,b}| \leq \text{SetRP}_{i_2}. \tag{13}$$

Constraint (14) states that for each $(i_1, i_2) \in \texttt{IretEdge}$, if $\exists b \in \texttt{Bank}.\text{RPVal}_{i_1,b} \neq \text{RPVal}_{i_2,b}$, then $\text{PopRP}_{i_1} = 1$:

$$\forall (i_1, i_2) \in \texttt{IretEdge}.\ \forall b \in \texttt{Bank}.\ |\text{RPVal}_{i_1,b} - \text{RPVal}_{i_2,b}| \leq \text{PopRP}_{i_1}. \tag{14}$$

In this case we do not have the option of introducing **srp** immediately before i_2; Figure 7 illustrates a scenario in which doing so results in incorrect behavior of the target program.

Finally, constraint (15) states that **srp** cannot be introduced immediately before i_{root}:

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

4.3.3 *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 each such instruction:

$$\forall (i, v_1, v_2) \in \texttt{Bin2Instr}.\ \forall b \in \texttt{Bank}.\ |\text{r}_{v_1,b} - \text{RPVal}_{i,b}| \leq \text{Bin2Cost}_i \tag{16}$$

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

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

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

$$\forall (i, v_1, v_2) \in \texttt{Bin1Instr}.\ \forall b \in \texttt{Bank}.\ |\text{r}_{v,b} - \text{RPVal}_{i,b}| \leq \text{Bin1Cost}_i. \tag{18}$$

Thus, for any $(i, v) \in \texttt{Bin1Instr}$, v will be in a working register whenever i is executed if $\text{Bin1Cost}_i = 0$.

The operand of any unary instruction in Z86E30's instruction set can be stored in a working or nonworking register, with the exception of the **djnz** *v, dst* instruction, in which the operand *v must* be stored in a working register. (Program memory addresses such as *dst* in **djnz** are not treated as operands. Hence, **djnz** qualifies as a unary instruction.) This condition is enforced by the following constraint:

$$\forall (i, v) \in \texttt{DjnzInstr}.\ \forall b \in \texttt{Bank}.\ r_{v,b} = \text{RPVal}_{i,b}. \tag{19}$$

4.4 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 `Bin1Instr` depends upon whether its operands are stored in working registers or not, and is characterized by the variables Bin2Cost and Bin1Cost, respectively. Also, the space cost of each of **srp**, **push RP**, and **pop RP** is 2 bytes. Thus, the objective function is

$$\sum_{(i,v_1,v_2)\in\texttt{Bin2Instr}} \text{Bin2Cost}_i \ + \sum_{(i,v)\in\texttt{Bin1Instr}} \text{Bin1Cost}_i \ +$$

$$\sum_{i\in\text{Instr}_{abs}} 2 * \text{SetRP}_i \ + \sum_{i\in\texttt{IretInstr}} 4 * \text{PopRP}_i.$$

5. CODE GENERATION

Given a solution to the ILP generated from the model of a ZIL program and the Exhaustive ILP formulation, we generate target Z86 assembly code as follows:

Number of:	serial	drop	rop	cturk	zturk	gturk
Instructions in ZIL program	209	555	564	735	883	949
Instructions in model/Nodes in CFG_{abs}	92	340	356	466	553	595
Edges in CFG_{abs}	174	681	1186	1101	1398	1531
Instructions in Bin2Instr	8	62	68	126	131	145
Instructions in Bin1Instr	42	112	120	144	150	158
RP-sensitive instructions	50	174	188	270	281	303
Instr in DjnzInstr	0	9	9	6	7	7
Variables	27	63	64	74	77	86
Procedures	8	27	25	33	46	48
Interrupt handlers	3	2	2	1	1	1

Fig. 8. Benchmark characteristics.

(1) If $v \in \mathtt{Var} - (\mathtt{PDV0} \cup \mathtt{PDV15})$ and $r_{v,b} = 1$, then we store v in (any register in) bank b. Constraint (8) ensures that there is a unique such b.

(2) If $(i, v_1, v_2) \in \mathtt{Bin2Instr}$, 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 $\mathrm{Bin2Cost}_i$ is 0 or 1, respectively.

(3) If $(i, v) \in \mathtt{Bin1Instr}$, then v is addressed using the 4-bit or 8-bit addressing mode, depending upon whether $\mathrm{Bin1Cost}_i$ is 0 or 1, respectively.

(4) If $(i, v) \in \mathtt{DjnzInstr}$, then the register allocated to v is addressed using the 4-bit addressing mode.

(5) If $i \in \mathtt{Instr} - \mathtt{Instr}_{abs}$ and $v \in \mathtt{Var}$ is an operand of i, then the register allocated to v is addressed using the 8-bit addressing mode.

(6) If $i \in \mathtt{Instr}_{abs}$, $\mathrm{SetRP}_i = 1$, and b is the bank such that $\mathrm{RPVal}_{i,b} = 1$, then we introduce **srp** b immediately before i. Constraint (12) ensures that there is a unique such b.

(7) If $i \in \mathtt{IretInstr}$ and $\mathrm{PopRP}_i = 1$, then we introduce **pop RP** immediately before i and **push RP** immediately before the first instruction of the interrupt handler to which i belongs.

(8) If $\mathrm{RPVal}_{i_{root},b} = 1$, then we replace i_{root} by **srp** b. Constraint (15) ensures that there is no **srp** instruction before i_{root}.

6. EXPERIMENTAL RESULTS

We have evaluated the performance of the Inexpensive and Exhaustive ILP formulations on a range of benchmark ZIL programs with respect to two criteria: the ILP solving time and the size of the generated code. Based on these results, we have designed the Selective ILP formulation (Section 6.3) that offers a good trade-off between ILP solving time and size of the generated code.

6.1 Benchmark Characteristics

For our experiments, we have used six proprietary microcontroller programs provided by Greenhill Manufacturing. Greenhill has over a decade of experience producing environmental control systems for agricultural needs. The programs were originally written in Z86 assembly; we have converted them to ZIL using an automatic reverse engineering tool [Palsberg and Wallace 2002]. Some characteristics of these programs are presented in Figure 8.

Number of:	technique	serial	drop	rop	cturk	zturk	gturk
Seconds to solve ILP	Inexpensive	0.08	4.3	8	51	56	190
	Selective	0.06	110	320	3300	4800	8900
	Exhaustive	0.09	4200	1700	86400	86400	86400
srp	Inexpensive	1	1	1	1	1	1
	Selective	2	9	7	15	12	15
	Exhaustive	2	11	11	14	15	19
push/pop RP	Selective	0	0	0	1	1	1
	Exhaustive	0	2	2	1	1	1
bytes of Z86 code	Upper bound	511	1346	1422	1804	2123	2287
	Inexpensive	483	1269	1337	1685	2001	2159
	Selective	477	1248	1315	1655	1958	2110
	Exhaustive	477	1239	1305	1641	1950	2109
	Lower bound	461	1172	1234	1536	1842	1984
	Handwritten	488	1248	1298	1536	1776	1901

Fig. 9. Measurements.

Model extraction abstracts away about 37% of the instructions in a typical benchmark ZIL program. The RP-sensitive instructions in each benchmark are precisely those instructions in `Bin2Instr` and `Bin1Instr`. About 30% of the instructions in a typical benchmark ZIL program are RP-sensitive instructions.

6.2 Measurements

Figure 9 presents the results of code-size-aware compilation using the Inexpensive, Selective, and Exhaustive ILP formulations. The ILP solving time 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. Moreover, for the Exhaustive ILP formulation, CPLEX was limited to use at most 1 day (86,400 s). It was unable to find an optimal solution to the Exhaustive ILPs for `cturk`, `zturk`, and `gturk` within that time limit, so the corresponding results are based on the potentially suboptimal solution that it was able to find within 1 day.

The upper (resp. lower) bounds on the bytes of Z86 code in Figure 9 denote the size of the Z86 code generated under the condition that no RP-manipulating instructions (**srp** and **push**/**pop RP**) are introduced, and all RP-sensitive instructions use 8-bit (resp. 4-bit) register addressing. These bounds are unrealistic in that it is impossible to generate, for a real-world ZIL program, Z86 code that satisfies the above condition. The bounds are provided to remind the reader that no register allocation technique can generate Z86 code as large as the upper bound or, more importantly, as small as the lower bound.

6.3 Assessment

It is evident from Figure 9 that ILP solving time increases prohibitively from Inexpensive to Exhaustive. The Inexpensive and Exhaustive techniques explore two extremes: Inexpensive allows only one **srp** to be introduced right at the start of

Number of **srp** at:	serial	drop	rop	cturk	zturk	gturk
START	2	8	8	7	10	12
END	0	0	0	3	2	3
MISC	0	3	3	4	3	4

Fig. 10. Locations in which **srp** was introduced by the Exhaustive technique: START and END denote entry and exit points, respectively, of routines, and MISC denotes arbitrary locations within routines.

the program, while Exhaustive allows **srp** to be introduced before *any* instruction in (the model extracted from) the program. To seek middle ground, we studied the locations at which **srp** was introduced by the Exhaustive technique for each of the six benchmark programs. Figure 10 summarizes our findings: more than 75% of the locations were the entry and exit points of routines. This provides the motivation for our Selective technique that allows **srp** to be introduced immediately before only those instructions at the entry and exit points of routines. Let $\texttt{ProcInstr} \subseteq \texttt{Instr}_{abs}$ be the set of all such instructions. Then, the Selective ILP formulation is identical to the Exhaustive ILP formulation with the following additional constraint:

$$\forall i \in \texttt{Instr}_{abs} - \texttt{ProcInstr}.\ \text{SetRP}_i = 0. \tag{20}$$

It is instructive to compare the sizes of the Z86 programs generated using the Selective technique with those of the corresponding handwritten Z86 programs (see Figure 9). The handwritten programs are slightly more compact because of two reasons:

—The handwritten programs use programming tricks (for example, jumps from one routine to instructions within another routine). ZIL forbids such tricks, even at the expense of code size, because they yield programs that are unreadable and unamenable to maintenance and reuse.

—The handwritten programs implicitly use mutable arrays. All the elements of an array are allotted registers in the same bank. As a result, an operation on all the elements of an array can be performed efficiently by looping over the registers in the corresponding bank. Since ZIL does not currently support mutable arrays, the reverse engineering tool converts each element of each such array into a variable and unrolls all loops, thereby bloating code size. Future work involves supporting mutable arrays in ZIL.

Figure 11 illustrates for each of the six benchmarks, the sizes of the Z86 programs generated using the Inexpensive, Selective, and Exhaustive techniques, the lower bound, and the size of the handwritten Z86 program, all normalized with respect to the upper bound. Inexpensive yields 5% space savings over the upper bound. Selective yields 2–3% space savings over Inexpensive. However, Exhaustive yields only 0.5% space savings over Selective. Moreover, CPLEX does not find the optimal solution to the Exhaustive ILPs for the largest three benchmarks but it does find the optimal solution to the Selective ILPs for all the

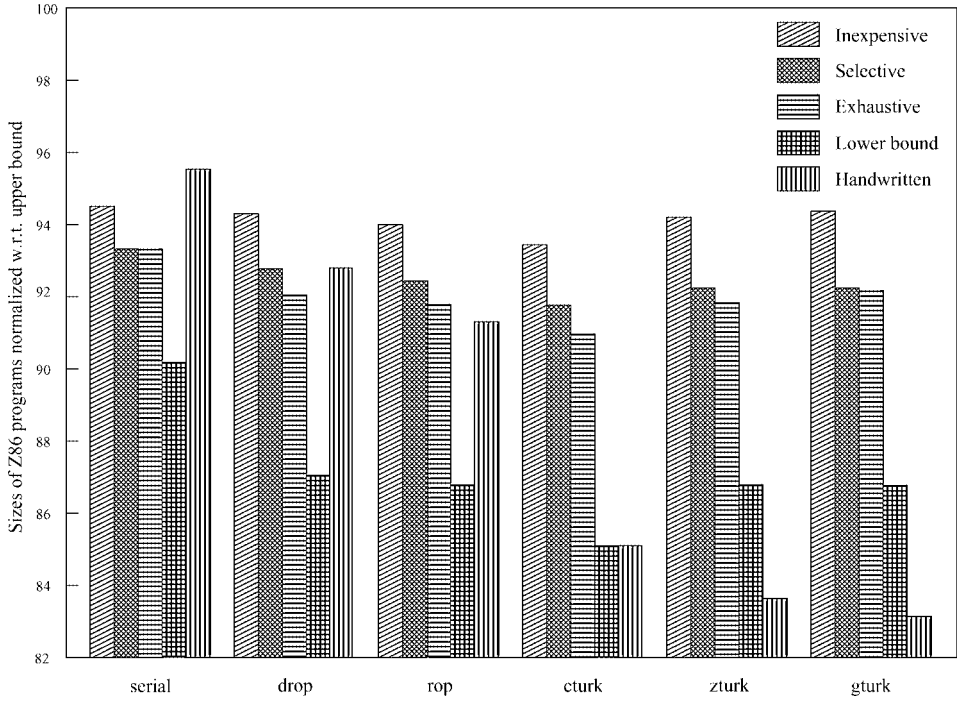

Fig. 11. Size of Z86 code normalized with respect to the upper bound.

benchmarks (see Figure 9). Selective therefore offers the best trade-off between ILP solving time and space savings.

7. CONCLUSION

We have investigated three approaches to code-size-aware compilation based on ILP. The Selective approach offers a good trade-off between ILP solving time and space savings. Since ILP is NP-complete, generating optimally sized target code for large programs can be prohibitively time consuming. We have employed an alternative methodology: an *exhaustive* ILP technique can be used once and for all to determine patterns in code generation. Next, a *selective* ILP technique can be formulated to take advantage of such patterns and generate code that is nearly as compact as carefully handwritten code in reasonable time.

APPENDIX

The grammar for ZIL is shown in Figure 12. A ZIL program consists of a `MAIN` part, procedures, and interrupt handlers. Each ZIL routine (procedure or interrupt handler) has a single entry point and a single exit point. In particular, a jump from one routine to an instruction within another routine is forbidden. Procedures may define formal parameters that may be passed by value or

```
            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" | "int[]"
         ProcDef ::= Label "(" (ParamList)? ")" ProcedureBlock
       ParamList ::= ParamDef ("," ParamDef)*
        ParamDef ::= Type ("&")? Id
      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 (ArgList)?
                   |  "preserveIMR" "{" ( Stmt )* ( Label ":" )? "}"
          LDExpr ::= "@" Id | Expr | "LABEL" Label
            Expr ::= "!" Expr | "(" Expr "&" Expr ")" | "(" Expr "|" Expr ")"
                   |  Prim
            Prim ::= Id | IntLiteral | ArrayReference
  ArrayReference ::= Id "[" IntLiteral "]" | Id "[" Id "]"
       LabelExpr ::= Label | "@" Variable
         ArgList ::= "(" Id ("," Id)* ")"
        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 | ArrayLiteral
       IntLiteral ::= <HEX_H> | <BIN_B> | <DEC_D>
       StrLiteral ::= <STRING_CONSTANT>
     ArrayLiteral ::= "{" IntLiteral ("," IntLiteral)* "}"
               Id ::= <IDENTIFIER>
```

Fig. 12. The ZIL grammar.

by reference. Formal parameters are considered to be local to the procedure, but may be passed to other procedures. Variables can be declared as global or as local to any routine. Additionally, ZIL has immutable integer arrays with Java-like syntax.

ACKNOWLEDGMENTS

The paper is a revised version of Naik and Palsberg [2002]. We thank Matthew Wallace for writing a ZIL interpreter and a Z86-to-ZIL reverse-engineering tool. We thank Dennis Brylow for help in numerous situations. We thank the anonymous referees for useful comments. Our research was supported by a National Science Foundation Information Technology Research award number 0112628.

REFERENCES

APPEL, A. AND GEORGE, L. 2001. Optimal spilling for CISC machines with few registers. In *Proceedings of PLDI'01, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 243–253.

AVISSAR, O., BARUA, R., AND STEWART, D. 2001. Heterogeneous memory management for embedded systems. In *Proceedings of CASES '01, International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*.

BRYLOW, D., DAMGAARD, N., AND PALSBERG, J. 2001. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, 47–56.

FOURER, R., GAY, D. M., AND KERNIGHAN, B. W. 1993. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press.

GOODWIN, D. W. AND WILKEN, K. D. 1996. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software—Practice & Experience 26*, 8 (Aug.), 929–965.

ILOG. *ILOG CPLEX 7.1 User's Manual*. See http://www.ilog.com.

KONG, T. AND WILKEN, K. D. 1998. Precise register allocation for irregular register architectures. In *Proceedings of MICRO '98, 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 297–307.

KREMER, U. 1997. Optimal and near-optimal solutions for hard compilation problems. *Parallel Processing Letters 7*, 4.

LEUPERS, R. AND MARWEDEL, P. 1996. Algorithms for address assignment in DSP code generation. In *Proceedings of ICCAD '96, International Conference on Computer Aided Design*, 109–112.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1996. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems 18*, 3 (May), 235–253.

LIBERATORE, V., FARACH-COLTON, M., AND KREMER, U. 1999. Evaluation of algorithms for local register allocation. 137–152.

MOTWANI, R., PALEM, K., SARKAR, V., AND REYEN, S. 1995. Combining register allocation and instruction scheduling. Tech. Rep. STAN-CS-TN-95-22, Department of Computer Science, Stanford University.

NAIK, M. AND PALSBERG, J. 2002. Compiling with code-size constraints. In *Proceedings of LCTES'02, Languages, Compilers, and Tools for Embedded Systems joint with SCOPES'02, Software and Compilers for Embedded Systems*, 120–129.

PALSBERG, J. AND WALLACE, M. 2002. Reverse engineering of real-time assembly code. Manuscript.

PARK, J., LEE, J., AND MOON, S. 2001. Register allocation for banked register file. In *Proceedings of LCTES '01, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, 39–47.

RAO, A. AND PANDE, S. 1999. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *Proceedings of PLDI '99, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 128–138.

RUTTENBERG, J., GAO, G. R., STOUTCHININ, A., AND LICHTENSTEIN, W. 1996. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of PLDI '96, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1–11.

SJÖDIN, J. AND VON PLATEN, C. 2001. Storage allocation for embedded processors. In *Proceedings of CASES '01, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 15–23.

STOUTCHININ, A. 1997. An integer linear programming model of software pipelining for the MIPS R8000 processor. In *Proceedings of PaCT'97, 4th International Conference on Parallel Computing Technologies*, 121–135.

SUDARSANAM, A. AND MALIK, S. 2000. Simultaneous reference allocation in code generation for dual data memory bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems 5*, 2 (Jan.), 242–264.

Zilog. *Z8 Microcontroller User's Manual*. See http://www.zilog.com.

Received September 2002; accepted June 2003