ILP-based Resource-aware Compilation

Jens Palsberg
UCLA

Mayur Naik
Stanford University

January 20, 2004

1 Introduction

Compilers are an important part of today's computational infrastructure because software is ever-increasingly written in high-level programming languages like C, C++, and Java. Early on, compilers were successful for desktop computing, and now they are also used for mission-critical software, trusted mobile code, and real-time embedded systems. The field of compiler design is driven by the advent of new languages, new run-time systems, and new computer architectures, and the need for compilers to be fast, to translate into efficient code, and to be correct. Even the smallest of bugs introduced by a compiler can lead to failures, security breaches, and crashes in an otherwise correct system.

This chapter focuses on compilers for embedded software. For embedded systems, predictability and resource-awareness are of greater importance than execution efficiency. For example, if a compiler is not aware of the size of the instruction store for generated code, opportunities to squeeze code into the available space can be lost. A programmer can often hand-optimize generated code, but this is tedious and error-prone, and does not scale to large systems. We present a snap-shot of the state-of-the-art and challenges faced by resource-aware compilers. We focus on a technique commonly used in resource-aware compilation, namely, *static analysis* based on *integer linear programming* (ILP).

A static analysis determines facts about a program that are useful for optimizations. One can present a static analysis in a variety of ways, including data-flow equations, set constraints, type constraints, or integer constraints. The use of integer constraints is an emerging trend in approaches to register allocation, instruction scheduling, code-size minimization, energy efficiency, and so forth. In general, integer constraints seem well-suited for building resource-aware compilers. The resources in question can be registers, execution time, code space, energy, or others that are in short supply.

Resource-aware compilation is particularly important for embedded systems where economic considerations often lead to the choice of a resource-impoverished processor. Integer constraints, in the form of ILPs, have been used to assist in the compilation of embedded software since the early 1990's. Satisfaction of an ILP is NP-complete, but there are many research results and tools concerned with solving ILPs efficiently. Off-the-shelf tools like CPLEX [1] make ILP well-suited for use in compilers because they provide analysis engines that need to be written once and for all. In effect, a compiler writer sees a clean separation

1

of "*what* the analysis is supposed to do" from "*how* the analysis is done." Once the analysis is specified, the ILP solving tool does the rest.

An ILP-based analysis is an instance of the more general notion of a constraint-based analysis. The idea of a constraint-based analysis is to do static program analysis in two steps: (1) generate constraints from the program text and (2) solve the constraints. For ILPs, a constraint is of the form $C_1 V_1 + \ldots + C_n V_n \sim C$ 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 $=$. A particularly popular subcase is the one in which all variables range over $\{0, 1\}$. Satisfiability of general ILPs and ILPs with just 0-1 variables are both NP-complete [16].

ILPs are quite different from type constraints or set constraints. Type constraints use variables that range over types, that is, finite or infinite trees, while set constraints use variables that range over sets. Integers have less structure than trees and sets but remain highly expressive. For a fair comparison of ILPs and set constraints, consider set constraints of the form $e \subseteq e'$ where the set expressions e, e' can use union, intersection, and complement, and variables that range over a set of constants $\{c_1, \ldots, c_n\}$. Satisfiability of such set constraints is NP-complete, just like satisfiability of ILPs. If X, Y range over subsets of $\{c_1, \ldots, c_n\}$, then the set constraint $X \subseteq Y$ can be translated into n integer linear constraints $x_1 \leq y_1, \ldots, x_n \leq y_n$ where $x_1, \ldots, x_n, y_1, \ldots, y_n$ are 0-1 variables, $x_i = 1$ represents that $c_i \in X$, and $y_i = 1$ represents that $c_i \in Y$. So, are ILPs merely a verbose way of formulating a certain form of set constraints? No! Integers are more convenient for specifying resource bounds, simply because such things as code size, number of registers, voltage levels, deadlines, etc. can be naturally phrased or represented as integers. Resource constraints therefore become integer constraints, rather than type constraints or set constraints.

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. The reasons include that (1) different optimizations like register allocation and instruction scheduling can be combined and solved within the same ILP framework, thereby avoiding the "phase ordering problem" [14, 8], (2) ILP guarantees optimality of the solution as opposed to heuristic methods [26], and (3) 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 [8].

Goodwin and Wilken [5] pioneered the use of ILP for register allocation, in the setting of processors with uniform register architectures. Kong and Wilken [7] use ILP for irregular register architectures like the IA-32. Appel and George [2] 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 [31] and Ruttenberg et al. [26] present an ILP framework for integrated register allocation and software pipelining in the MIPS R8000 microprocessor. Liberatore et al. [10] 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 sub-optimal. Sjödin and Platen [30] model multiple address spaces of a certain processor using ILP, and Avissar, Barua, and Stewart [3] use ILP to optimally allocate global and stack data among different heterogeneous memory modules.

2

2   Examples

In this section we present the formulation of four recent ILP-based analyses in a uniform format for a simple example language. The four analyses are concerned with instruction scheduling, energy efficiency, code-size minimization, and register allocation. Casting the analyses in a common framework makes it easier to understand fundamental similarities and differences. For each of the four analyses, we specify the following items.

Processor: The key resource constraint of the intended processor.

Source Program: Information about the source program that we assume is given to the ILP-based analysis, such as *liveness information.* Liveness information expresses, for each program point, which variables will be needed later in the computation.

Target Program: The required format of the target program, including special instructions provided by the target language.

Problem: The optimization problem that the ILP-based analysis is intended to solve.

0-1 Variables: All four analyses use only 0-1 variables.

Constraints: We use ILPs. In this proposal, we will use constraints of just four forms:

$$
\begin{aligned}
V_1 + \ldots + V_n &= 1 \\
V_1 + \ldots + V_n &\leq C \\
|V_1 - V_2| &\leq V_3 \\
V_1 + V_2 &= V_3 + V_4.
\end{aligned}
$$

where $V_1, \ldots, V_n$ are variables that range over $\{0, 1\}$, and C is an integer constant. It is easy to see that all such constraints can be encoded using the general format of ILPs.

Objective Function: The analysis will try to minimize the value of the objective function.

Code Generation: When a solution to the ILP has been found, code can be generated.

Example: We have a running example which is a little program in our example language. For each of the four optimizations, we show the code generated for the example program.

A program in our example language is an instruction list $s_1, ..., s_n$. Each instruction is an assignment of the form $x := c$ or $x \mathrel{+}= y$, where x, y range over integer variables and c ranges over integer constants. Our example language can be viewed as a means for expressing basic blocks in a conventional language like C. In the future, we hope to extend the language.

A program in the example language is specified using sets. $\mathbf{P}$ is the set of program points, where a point lies between two successive instructions or precedes s_1 or follows s_n. `Pair` $\subseteq (\mathbf{P} \times \mathbf{P})$ is the set of pairs of adjacent points and `Succ` $= \{ (p_1, p_2, p_3) \mid (p_1, p_2) \in$ `Pair` $\wedge (p_2, p_3) \in$ `Pair`$\}$. $\mathbf{V}$ is the set of integer variables used in the program. $\mathbf{U} \subseteq (\mathbf{P} \times \mathbf{P} \times \mathbf{V})$ is the set of triples (p_1, p_2, x) such that $(p_1, p_2) \in$ `Pair` and the instruction between p_1 and p_2 is $x := c$, and $\mathbf{B} \subseteq (\mathbf{P} \times \mathbf{P} \times \mathbf{V} \times \mathbf{V})$ is the set of quadruples (p_1, p_2, x, y) such that $(p_1, p_2) \in$ `Pair` and the instruction between p_1 and p_2 is $x \mathrel{+}= y$. Additional sets specifying, for instance, data dependencies or liveness information, will be defined when required.

For the example program in Figure 1(a), $\mathbf{P} = \{1, ..., 7\}$, `Pair` $= \{(i, i + 1) \mid i \in [1..6]\}$, $\mathbf{V} = \{u, v, x, y\}$, $\mathbf{U} = \{(1, 2, u), (2, 3, v), (4, 5, x), (5, 6, y)\}$, and $\mathbf{B} = \{(3, 4, v, u), (6, 7, y, x)\}$.

$u := c_u$ $u := c_u,\ v := c_v$ **svl** 1   **srp** 1 $u := c_u$ **srp** 2
$v := c_v$ $x := c_x,\ y := c_y$ $u := c_u$ $r1 := c_u$ $r := c_v$ **svl** 1
$v += u$ $v += u,\ y += x$ $v := c_v$ $r2 := c_v$ $r += u$ $R11 := c_u$
$x := c_x$ $v += u$ $r2 += r1$ $x := c_x$ $R12 := c_v$
$y := c_y$ **svl** 2   **srp** 2 $r := c_y$ $R12 += R11$
$y += x$ $x := c_x$ $r1 := c_x$ $r += x$ $r1 := c_x$
                         $y := c_y$ $r2 := c_y$ **svl** 2
                         $y += x$ $r2 += r1$ $r2 := c_y$
                                                 $r2 += r1$

(a)   (b)   (c)   (d)   (e)   (f)

Figure 1: Example Program

2.1   Instruction Scheduling

This section presents the ILP-based analysis for instruction scheduling by Wilken, Liu, and Heffernan [37], recast for our example language.

Processor: The processor can schedule R instructions simultaneously.

Source Program: The source program is provided in the form of the standard sets and the set $\texttt{Depn} \subseteq (\mathbf{P} \times \mathbf{P} \times \mathbf{P} \times \mathbf{P} \times \mathbb{N})$ consisting of 5-tuples (p_1, p_2, p_3, p_4, W) such that $(p_1, p_2) \in \texttt{Pair}$, $(p_3, p_4) \in \texttt{Pair}$, and the instruction between p_1 and p_2 can begin executing no earlier than W cycles after the instruction between p_3 and p_4 has begun executing.

Target Program: The target program is a list of groups of instructions (as opposed to a list of instructions): Each group consists of at most R instructions all of which are scheduled simultaneously. The schedule length of a target program is the length of the list. We are given (i) a lower bound L on the schedule length and (ii) a target program with schedule length U possibly computed using a heuristic.

Problem: Generate a target program that has the minimum schedule length.

The ILP formulation (described below) is parameterized by the schedule length (M). If $U = L$, the schedule is optimal, and the formulation is not instantiated. If $U > L$, the formulation is instantiated with schedule length $U - 1$. If the resulting ILP is infeasible, the schedule of length U is optimal. Otherwise, a schedule of length $U - 1$ was found, and the formulation is instantiated with schedule length $U - 2$. This procedure is repeated until a schedule of minimum length is found.

0-1 Variables: Variable x is defined such that for each $(p_1, p_2) \in \texttt{Pair}$ and each cycle $i \in [1..M]$, $x_{p_1, p_2, i} = 1$ iff the instruction between p_1 and p_2 is scheduled in cycle i.

Constraints: Each instruction must be scheduled in exactly one of the M cycles and at most R instructions can be scheduled simultaneously.

$$\forall (p_1, p_2) \in \texttt{Pair}. \ \sum_{i=1}^{M} x_{p_1, p_2, i} = 1$$

$$\forall i \in [1..M]. \ \sum_{(p_1, p_2) \in \texttt{Pair}} x_{p_1, p_2, i} \leq R$$

4

For each $(p_1, p_2) \in$ Pair, the cycle in which the instruction between p_1 and p_2 is scheduled is given by:

$$\sum_{i=1}^{M} i \ x_{p_1,p_2,i}$$

Using this formula, a constraint is expressed for each $(p_1, p_2, p_3, p_4, W) \in$ Depn to ensure that the instruction between p_1 and p_2 is scheduled no earlier than W cycles after the instruction between p_3 and p_4 has been scheduled.

$$\forall (p_1, p_2, p_3, p_4, W) \in \text{Depn}. \ \sum_{i=1}^{M} i \ x_{p_3,p_4,i} + W \ \leq \ \sum_{i=1}^{M} i \ x_{p_1,p_2,i}$$

Code Generation: The target program generated from a solution to the ILP obtained from the above ILP formulation and the source program is a list of groups of instructions, namely, if $(p_1, p_2) \in$ Pair and i is the cycle such that $x_{p_1,p_2,i} = 1$, then the instruction between p_1 and p_2 in the source program is scheduled in the ith group in the target program.

Example: Suppose the example program has four dependencies, namely, that $v \mathrel{+}= u$ must begin executing no earlier than 1 cycle after $u := c_u$ and $v := c_v$ have begun executing, and that $y \mathrel{+}= x$ must begin executing no earlier than 1 cycle after $x := c_x$ and $y := c_y$ have begun executing, *i.e.*, Depn= { (3, 4, 1, 2, 1), (3, 4, 2, 3, 1), (6, 7, 4, 5, 1), (6, 7, 5, 6, 1) }. With $R = 2$, the optimal solution to the ILP obtained from the above ILP formulation and the example program yields a target program with schedule length 3. The target program generated using one such solution is shown in Figure 1(b). Note that the program respects the stated dependencies: $v \mathrel{+}= u$ is scheduled 2 cycles after $u := c_u$ and $v := c_v$, and $y \mathrel{+}= x$ is scheduled 1 cycle after $x := c_x$ and $y := c_y$.

2.2 Energy Efficiency

This section presents the ILP-based analysis for energy efficiency by Saputra et al. [27], recast for our example language.

Processor: The processor has N voltage levels.

Source Program: The source program is provided in the form of the standard sets and, for each $(p_1, p_2) \in$ Pair and $i \in [1..N]$, $E_{p_1,p_2,i}$ (resp. $T_{p_1,p_2,i}$) is the energy consumption (resp. execution time) of the instruction between p_1 and p_2 when voltage level i is used. Also, the overall execution time must not exceed a deadline D.

Target Program: The target language differs from the source language in that it provides a special instruction **svl** i ("set voltage level to i"), with execution time C_t and energy consumption C_e, to change the voltage level at any program point.

Problem: Generate a target program that consumes minimum energy.

0-1 Variables:

- Variable VLVal is defined such that for each $(p_1, p_2) \in$ Pair and each $i \in [1..N]$, $\text{VLVal}_{p_1,p_2,i} = 1$ iff voltage level i is selected for the instruction between p_1 and p_2.

- Variable SetVL is defined such that for each $(p_1, p_2, p_3) \in$ Succ, $\text{SetVL}_{p_2} = 1$ if the voltage level changes at p_2.

Constraints: Exactly one voltage level should be selected for each instruction.

$$\forall (p_1, p_2) \in \texttt{Pair}. \ \sum_{i=1}^{N} \text{VLVal}_{p_1,p_2,i} = 1$$

For each $(p_1, p_2, p_3) \in \texttt{Succ}$, if the voltage level at the instruction between p_1 and p_2 differs from that at the instruction between p_2 and p_3, then $\text{SetVL}_{p_2} = 1$.

$$\forall (p_1, p_2, p_3) \in \texttt{Succ}. \ \forall i \in [1..N]. \ |\text{VLVal}_{p_1,p_2,i} - \text{VLVal}_{p_2,p_3,i}| \ \leq \ \text{SetVL}_{p_2}$$

The overall execution time must not exceed D.

$$\sum_{(p_1,p_2)\in \texttt{Pair}} \sum_{i=1}^{N} T_{p_1,p_2,i} \, \text{VLVal}_{p_1,p_2,i} \ + \sum_{(p_1,p_2,p_3)\in \texttt{Succ}} C_t \, \text{SetVL}_{p_2} \ + \ C_t \ \leq \ D \qquad (1)$$

The term C_t on the *l.h.s.* of the above constraint accounts for the execution time of an **svl** instruction introduced at the start of the target program (see code generation below).

Objective Function: Minimize the energy consumption of the target program:

$$\sum_{(p_1,p_2)\in \texttt{Pair}} \sum_{i=1}^{N} E_{p_1,p_2,i} \, \text{VLVal}_{p_1,p_2,i} \ + \sum_{(p_1,p_2,p_3)\in \texttt{Succ}} C_e \, \text{SetVL}_{p_2}$$

Code Generation: The target program generated from a solution to the ILP obtained from the above ILP formulation and the source program $s_1, ..., s_n$ is identical to the source program with the following modifications:

- If $(p_1, p_2, p_3) \in \texttt{Succ}$, $\text{SetVL}_{p_2} = 1$, and i is the voltage level such that $\text{VLVal}_{p_2,p_3,i} = 1$, then we introduce instruction **svl** i at p_2.

- If $(p_1, p_2) \in \texttt{Pair}$ where p_1 is the program point before s_1 and i is the voltage level such that $\text{VLVal}_{p_1,p_2,i} = 1$, then we introduce instruction **svl** i at p_1.

Example: With $N = 2$, $D = 38$, $C_e = 7$, $C_t = 2$, and the energy consumption and execution time of the unary and binary instructions at the two voltage levels given below:

Voltage Level:	1	2
instruction in **U**	10	20
instruction in **B**	15	30

Energy Consumption

Voltage Level:	1	2
instruction in **U**	6	3
instruction in **B**	10	5

Execution Time

The optimal solution to the ILP obtained from the above ILP formulation and the example program yields a target program whose energy consumption is 119 units. The target program generated using one such solution is shown in Figure 1(c). Note that the program has an overall execution time of 37 cycles and thereby meets the specified deadline of 38 cycles.

2.3   Code-Size Minimization

This section presents the ILP-based analysis for code-size minimization by Naik and Palsberg [15], recast for our example language.

Processor: The processor has a banked register file of N banks and M registers per bank, and a Register Pointer (RP) indicating the "working bank."

Source Program: The source program is provided in the form of the standard sets.

Target Program: The target language differs from the source language in two respects: (1) it provides a special instruction **srp** b ("set RP to bank b"), occupying 2 bytes, to change the working bank at any program point and (2) it uses registers instead of variables. A variable allotted register d in bank b is denoted rd or Rbd, depending upon whether b is the working bank or not, respectively. The target instruction corresponding to $x := c$ occupies 2 or 3 bytes depending upon whether x is allotted a register in the working bank or not, respectively, and the target instruction corresponding to $x \mathrel{+}= y$ occupies 2 or 3 bytes depending upon whether x and y are allotted registers in the working bank or not, respectively.

Problem: Generate a target program that occupies minimum space.

0-1 Variables:

- Variable r is defined such that for each $v \in \mathbf{V}$ and $b \in [1..N]$, $r_{v,b} = 1$ iff v is allotted a register in bank b.

- Variable RPVal is defined such that for each $(p_1, p_2) \in \mathtt{Pair}$ and $b \in [1..N]$, $\mathrm{RPVal}_{p_1,p_2,b} = 1$ if the value of RP at the instruction between p_1 and p_2 is b.

- Variable SetRP is defined such that for each $(p_1, p_2, p_3) \in \mathtt{Succ}$, $\mathrm{SetRP}_{p_2} = 1$ if the value of RP changes at p_2.

- Variable UCost_{p_1,p_2} is defined such that for each $(p_1, p_2, v) \in \mathbf{U}$, $\mathrm{UCost}_{p_1,p_2} = 1$ if the value of RP at the instruction between p_1 and p_2 is not the bank in which the register for v is allotted.

- Variable BCost_{p_1,p_2} is defined such that for each $(p_1, p_2, v_1, v_2) \in \mathbf{B}$, $\mathrm{BCost}_{p_1,p_2} = 1$ if the value of RP at the instruction between p_1 and p_2 is not the bank in which the register for v_1 or v_2 (or both) is allotted.

Constraints: A variable must be stored in exactly one bank, the total number of variables must not exceed the total number of registers, and RP must be set to exactly one bank at every instruction.

$$\forall v \in \mathbf{V}. \quad \sum_{b \in [1..N]} r_{v,b} = 1$$

$$\forall b \in [1..N]. \quad \sum_{v \in \mathbf{V}} r_{v,b} \leq M$$

$$\forall (p_1, p_2) \in \mathtt{Pair}. \quad \sum_{b=1}^{N} \mathrm{RPVal}_{p_1,p_2,b} = 1$$

For each $(p_1, p_2, p_3) \in$ Succ, if the value of RP at the instruction between p_1 and p_2 differs from that at the instruction between p_2 and p_3, then $\text{SetRP}_{p_2} = 1$.

$$\forall (p_1, p_2, p_3) \in \text{Succ}. \ \forall b \in [1..N]. \ |\text{RPVal}_{p_1,p_2,b} - \text{RPVal}_{p_2,p_3,b}| \ \leq \ \text{SetRP}_{p_2}$$

Any instruction in **U** occupies 2 or 3 bytes depending upon whether the operand is stored in the working bank or not, respectively. The following constraint characterizes the space cost of each such instruction.

$$\forall (p_1, p_2, v) \in \mathbf{U}. \ \forall b \in [1..N]. \ |\text{r}_{v,b} - \text{RPVal}_{p_1,p_2,b}| \leq \text{UCost}_{p_1,p_2}$$

Any instruction in **B** occupies 2 or 3 bytes depending upon whether both operands are stored in the working bank or not, respectively. The following constraints characterize the space cost of each such instruction.

$$\forall (p_1, p_2, v_1, v_2) \in \mathbf{B}. \ \forall b \in [1..N]. \ |\text{r}_{v_1,b} - \text{RPVal}_{p_1,p_2,b}| \leq \text{BCost}_{p_1,p_2}$$
$$\forall (p_1, p_2, v_1, v_2) \in \mathbf{B}. \ \forall b \in [1..N]. \ |\text{r}_{v_2,b} - \text{RPVal}_{p_1,p_2,b}| \leq \text{BCost}_{p_1,p_2}$$

Objective Function: Minimize the space occupied by the target program:

$$\sum_{(p_1,p_2,v)\in\mathbf{U}} \text{UCost}_{p_1,p_2} \ + \sum_{(p_1,p_2,v_1,v_2)\in\mathbf{B}} \text{BCost}_{p_1,p_2} \ + \sum_{(p_1,p_2,p_3)\in\text{Succ}} 2\,\text{SetRP}_{p_2}$$

Code Generation: The target program generated from a solution to the ILP obtained from the above ILP formulation and the source program $s_1, ..., s_n$ is identical to the source program with the following modifications:

- If $v \in \mathbf{V}$ and b is the bank such that $r_{v,b} = 1$, then v is allotted a unique register in bank b.

- If $(p_1, p_2, v) \in \mathbf{U}$ and v is allotted register d in bank b, then the instruction between p_1 and p_2 is denoted $rd := c$ or $Rbd := c$, depending upon whether UCost_{p_1,p_2} is 0 or 1, respectively.

- If $(p_1, p_2, v_1, v_2) \in \mathbf{B}$ and v_1, v_2 are allotted registers d_1, d_2 in banks b_1, b_2, respectively, then the instruction between p_1 and p_2 is denoted $rd_1 \ += \ rd_2$ or $Rb_1d_1 \ += \ Rb_2d_2$, depending upon whether BCost_{p_1,p_2} is 0 or 1, respectively.

- If $(p_1, p_2, p_3) \in \text{Succ}$, $\text{SetRP}_{p_2} = 1$, and b is the bank such that $\text{RPVal}_{p_2,p_3,b} = 1$, then we introduce instruction **srp** b at p_2.

- If $(p_1, p_2) \in \text{Pair}$ where p_1 is the program point before s_1 and b is the bank such that $\text{RPVal}_{p_1,p_2,b} = 1$, then we introduce instruction **srp** b at p_1.

Example: With $N = M = 2$, the optimal solution to the ILP obtained from the above ILP formulation and the example program yields a target program that occupies 16 bytes. The target program generated using one such solution is shown in Figure 1(d) (variables u and v are stored in the first and second registers of bank 1 while variables x and y are stored in the first and second registers of bank 2).

8

2.4   Register Allocation

This section presents the ILP-based analysis for register allocation by Appel and George [2], recast for our example language.

Processor: The processor consists of K registers.

Source Program: The source program is provided in the form of the standard sets and `Live` $\subseteq (\mathbf{P} \times \mathbf{V})$, the set of pairs (p, v) such that variable v is live at program point p, and `Copy` $= \{ (p_1, p_2, v) \mid (p_1, p_2) \in$ `Pair` $\wedge (p_1, v) \in$ `Live` $\wedge (p_2, v) \in$ `Live` $\}$.

Target Program: The target language differs from the source language in two respects: (1) it uses registers or memory addresses instead of variables: a variable in register d is denoted rd and a variable in memory is denoted by the variable name itself, and (2) it provides special instructions **ld** rd, x ("load variable from memory address x to register d") and **st** x, rd ("store register d into memory address x"), each occupying 3 bytes.

The cost of loading (resp. storing) a variable is C_l (resp. C_s) cycles. The cost of fetching and decoding one instruction byte is C_i cycles. The target instruction corresponding to $x := c$ occupies 1 or 2 bytes depending upon whether x is a register or memory address, respectively. The target instruction corresponding to $x \mathrel{+}= y$ occupies 2, 3, or 4 bytes depending upon whether both x and y are registers, one is a register and the other is a memory address, or both are memory addresses, respectively.

Problem: Generate a target program that executes in minimum time.

0-1 Variables: Variables r, m, l, s are defined such that for each $(p, v) \in$ `Live` we have:

$$
\begin{aligned}
r_{p,v} = 1 \quad &\text{iff} \quad v \text{ arrives at } p \text{ in a register and departs } p \text{ in a register.} \\
m_{p,v} = 1 \quad &\text{iff} \quad v \text{ arrives at } p \text{ in memory and departs } p \text{ in memory.} \\
l_{p,v} = 1 \quad &\text{iff} \quad v \text{ arrives at } p \text{ in memory and departs } p \text{ in a register.} \\
s_{p,v} = 1 \quad &\text{iff} \quad v \text{ arrives at } p \text{ in a register and departs } p \text{ in memory.}
\end{aligned}
$$

Constraints: For each $(p, v) \in$ `Live`, v must arrive at p either in a register or in memory and, likewise, depart p either in a register or in memory.

$$\forall (p, v) \in \text{Live. } r_{p,v} + m_{p,v} + l_{p,v} + s_{p,v} = 1$$

Since registers are scarcer than memory, all the stores will be performed before all the loads at each program point. Therefore, for each $p \in \mathbf{P}$, the sum of (i) the variables stored at p (*i.e.*, variables arriving at p in registers and departing p in memory) and (ii) the variables arriving at p in registers and departing p in registers must not exceed K.

$$\forall p \in \mathbf{P}. \sum_{(p,v) \in \text{Live}} s_{p,v} + r_{p,v} \leq K$$

If variable v is live at successive program points p_1 and p_2, then it either departs p_1 in a register (resp. memory) and arrives at p_2 in a register (resp. memory). If it departs p_1 in a register, then either it must have already been in a register ($r_{p_1,v} = 1$), or it must have been loaded at p_1 ($l_{p_1,v} = 1$). Likewise, if it arrives at p_2 in a register, then either it must continue in a register ($r_{p_2,v} = 1$), or it must be stored at p_1 ($s_{p_2,v} = 1$).

$$\forall (p_1, p_2, v) \in \text{Copy. } l_{p_1,v} + r_{p_1,v} = s_{p_2,v} + r_{p_2,v}$$

Objective Function: Minimize the execution time of the target program:

$$\sum_{(p,v)\in\texttt{Live}} \left((C_l + 3\,C_i)\,l_{p,v} + (C_s + 3\,C_i)\,s_{p,v}\right) + \sum_{(p_1,p_2,v)\in\mathbf{U}} (C_s + C_i)\,(m_{p_2,v} + l_{p_2,v}) +$$

$$\sum_{(p_1,p_2,v_1,v_2)\in\mathbf{B}} \left((C_l + C_i)\,(m_{p_1,v_1} + s_{p_1,v_1}) + (C_l + C_s + C_i)\,(m_{p_1,v_2} + l_{p_1,v_2})\right)$$

The first component accounts for the cost of each 3-byte load and store instruction that is introduced. The second component accounts for the "overhead" cost of each unary instruction $x := c$, namely, if x is in memory, it accounts for the cost to store x and the cost to fetch and decode an extra byte of the instruction. The third component accounts for the "overhead" cost of each binary instruction $x \mathrel{+}= y$, namely, (i) if y is in memory, it accounts for the cost to load y and the cost to fetch and decode an extra byte of the instruction, and (ii) if x is in memory, it accounts for the cost to load *and* store x and the cost to fetch and decode another extra byte of the instruction.

Code Generation: The target program generated from a solution to the ILP obtained from the above ILP formulation and the source program is identical to the source program with the following modifications (performed in that order):

- If $(p, v) \in \texttt{Live}$ and $l_{p,v} = 1$, then v is allotted a unique register (say d) and instruction **ld** rd, v is introduced at p.

- If $(p, v) \in \texttt{Live}$ and $s_{p,v} = 1$, then v was allotted some register (say d) and instruction **st** v, rd is introduced at p.

- If $(p_1, p_2, v) \in \mathbf{U}$, then $(p_1, v) \notin \texttt{Live}$ and $(p_2, v) \in \texttt{Live}$, and there are two cases dictating how the instruction between p_1 and p_2 must be denoted:

  If $r_{p_2,v} + s_{p_2,v} = 1$, then v is allotted a unique register (say d) and the instruction is denoted $rd := c$. Else, $m_{p_2,v} + l_{p_2,v} = 1$, and the instruction is denoted $v := c$.

- If $(p_1, p_2, v_1, v_2) \in \mathbf{B}$, then $(p_1, v_1) \in \texttt{Live}$ and $(p_1, v_2) \in \texttt{Live}$, and there are four cases dictating how the instruction between p_1 and p_2 is denoted:

  If $r_{p_1,v_1} + l_{p_1,v_1} = 1$, then v_1 was allotted some register (say d_1) before the instruction. If $r_{p_1,v_2} + l_{p_1,v_2} = 1$, then v_2 was allotted some register (say d_2) before the instruction. Then, the instruction is denoted one of $rd_1 \mathrel{+}= rd_2$, $rd_1 \mathrel{+}= v_2$, $v_1 \mathrel{+}= rd_2$, and $v_1 \mathrel{+}= v_2$, depending upon whether v_1 and v_2 were both allotted registers, only v_1 was allotted a register, only v_2 was allotted a register, or neither v_1 nor v_2 was allotted a register, respectively.

Example: With $K = 1$, $C_i = C_l = C_s = 2$, $\texttt{Live} = \{(2, u), (3, u), (3, v), (5, x), (6, x), (6, y)\}$, and $\texttt{Copy} = \{\}$, the optimal solution to the ILP obtained from the above ILP formulation and the example program yields a target program with execution time 16 cycles. The target program generated using one such solution is shown in Figure 1(e).

3   Open Problems

In this section we discuss three possible research directions. First, how can we design combinations of ILP-based analyses? This would, for instance, enable combining the phases of a compiler, thereby solving the phase-ordering problem. Second, how can we prove the correctness of ILP-based analyses, including combinations of such analyses? Third, how can we prove equivalence results that clarify the relationships between ILP-based analysis, type-based analysis, and set-based analysis? This would be a step towards a unified framework and a better understanding of when each of the approaches is the best fit for a given problem.

3.1   Combination

Performing different ILP-based transformations sequentially on a source program might result in the violation of certain constraints on the resource usage of the target program. For instance, consider the ILP formulations for energy efficiency (Section 2.2) and code-size minimization (Section 2.3), denoted I_E and I_S, respectively. The I_E-based transformation, when performed in isolation, ensures that the target program will consume the least energy. However, performing the I_S-based transformation following the I_E-based transformation does not ensure this. Likewise, the I_S-based transformation, when performed in isolation, ensures that the target program will occupy the least space. However, performing the I_E-based transformation following the I_S-based transformation does not ensure this.

   Precise control over both the energy usage and the space usage of the target program can be achieved by a single transformation based on an ILP formulation that is a combination of I_E and I_S, denoted I_{ES}. The constraints in I_{ES} are those in I_E and I_S, except that constraint (1) which constrains the overall execution time of the target program must also account for the execution time (say C_t) of each **srp** instruction that is introduced:

$$\sum_{(p_1,p_2)\in \texttt{Pair}} \sum_{i=1}^{N} T_{p_1,p_2,i} \, \text{VLVal}_{p_1,p_2,i} \; + \sum_{(p_1,p_2,p_3)\in \texttt{Succ}} C_t \left(\text{SetVL}_{p_2} + \text{SetRP}_{p_2} \right) \; + \; 2\,C_t \; \leq \; D$$

However, there is no one obvious way of dealing with multiple objective functions. Williams [38] suggests that one approach is to take a suitable linear combination of the objective functions. We will illustrate this approach by choosing the objective function of I_{ES} as the sum of the objective functions of I_E and I_S. Let the energy consumption of the **srp** instruction be C_e and the space occupied by the **svl** instruction be 2 bytes. The energy usage E and the space usage S of the target program are:

$$E = \sum_{(p_1,p_2)\in \texttt{Pair}} \sum_{i=1}^{N} E_{p_1,p_2,i} \, \text{VLVal}_{p_1,p_2,i} \; + \sum_{(p_1,p_2,p_3)\in \texttt{Succ}} C_e \left(\text{SetVL}_{p_2} + \text{SetRP}_{p_2} \right) \; + \; 2\,C_e$$

$$S = \sum_{(p_1,p_2,v)\in \mathbf{U}} \text{UCost}_{p_1,p_2} + \sum_{(p_1,p_2,v_1,v_2)\in \mathbf{B}} \text{BCost}_{p_1,p_2} + \sum_{(p_1,p_2,p_3)\in \texttt{Succ}} 2 \left(\text{SetRP}_{p_2} + \text{SetVL}_{p_2} \right) \; + \; 4$$

Then, the objective function of I_{ES} is $E + S$.

   The optimal solution to the ILP obtained from I_{ES} and the example program in Figure 1(a) yields the target program in Figure 1(f) which has space usage 21 bytes and energy usage 116 units.

3.2 Correctness

Until now, there have been only a few results on the correctness properties of ILP-based analyses. This is in marked contrast to type-based analyses and set-based analyses, for which many theoretical results are available. Key correctness properties include soundness, preservation, and combination, that is: (1) is the analysis sound with respect to a formal semantics? (2) is the analysis preserved after program transformations and optimizations? and (3) can analyses be combined in ways that preserve basic properties of the program? Foundational results about the correctness of ILP-based analyses can lead to improved compiler technology, quicker debugging of compilers, fewer bugs and increased confidence in generated code, principles for developing new analyses, increased understanding of how to combine analyses, and ILP-based code certification, in the spirit of proof-carrying code, typed assembly language, and Java bytecode verification.

An approach to correctness can take its starting point in the well-understood foundations of type-based analysis and set-based analysis. First, there are results on *soundness*, that is, the analysis is sound with respect to a formal semantics. Soundness is the key lemma for proving correctness of the optimization that is enabled by the analysis. For type systems, a typical soundness theorem reads:

> *A well-typed program cannot go wrong.*

This is a safety result: bad things cannot happen. For set-based analyses, a typical soundness theorem reads, informally:

> *If an expression e can produce a value v, then the set-based analysis of e gives a set which approximates v.*

Again, this is a safety result: unexpected things cannot happen. Milner [12] proved the first type soundness theorem, Sestoft [28, 29] proved that a set-based control-flow analysis is sound with respect to call-by-name and call-by-value semantics, and Palsberg [20] proved that the same analysis is sound for arbitrary beta-reduction, a proof that was later improved by Wand and Williamson [35]. A well-understood proof technique for type soundness [17, 39] uses a lemma that is usually known as Type Preservation or Subject Reduction. Intuitively, such lemmas state that if a program type checks and it takes one step of computation using a rewriting rule, then the resulting program type checks. A similar proof technique works for proving soundness for set-based analysis: Palsberg [20] demonstrated that a set-based analysis is preserved by one step of computation. Some soundness results have been automatically checked, *e.g.*, Nipkow and Oheimb's type soundness proof for a substantial subset of Java [19]. Among the correctness results for optimizations enabled by a program analysis is that of Wand and Steckler [34] for set-based closure conversion.

The *preservation* of typability and set-based analysis may go beyond merely steps of computation. In particular, the field of property-preserving compilation is concerned with designing and implementing compilers where each phase preserves properties established for the source program. Intuitively, the goal is to achieve results of the form:

> *If a program has a property, then the compiled program also has that property.*

Meyer and Wand [11] have shown that typability is preserved by CPS transformation, and Damian and Danvy [4] and Palsberg and Wand [25] have shown that a set-based flow analysis is preserved by CPS transformation. Examples of end-to-end type-preserving compilers with proofs include the TIL compiler of Tarditi et al. [33] and the compiler from System F to typed assembly language of Morrisett, Walker, Crary, and Glew [13].

Types and set-based analysis can be *combined* into so-called flow types. The idea is to use types that are annotated with sets. Flow types have been studied by Tang and Jouvelot [32], Heintze [6], Wells, Dimock, Muller, and Turbak [36], and others, who show how to prove soundness for various kinds of flow types. Correctness results for other kinds of annotated-type systems have been presented by Nielson, Nielson, and Hankin [18]. One can go further and establish the equivalence of type systems and set-based flow analysis [23, 6, 21, 24]. In these cases, a correctness result for one side can often be transferred easily to the other side. Lerner, Grove, and Chambers [9] combined five dataflow analyses into one analysis, thereby overcoming the phase-ordering problem. Their compiler produces results that are at least as precise as iterating the individual analyses while compiling at least five times faster.

It remains to be seen whether the proof techniques used for type-based and set-based analysis can also be applied to ILP-based analysis.

3.3   Relationships with other Approaches

For object-oriented and functional languages, most static analyses are based on a set-based flow analysis. This leads to the suggestion that resource-aware compilation for, say, Java can best be done using a combination of set-based analysis and ILP-based analysis. This is particularly interesting if the two kinds of constraints interact in nontrivial ways. Currently, there is no practical method for combining these; set-based analysis is usually formulated using set constraints and solved using one kind of constraint solver, while ILPs have an entirely different kind of constraint solver. Should the constraints be solved using a joint constraint solver or should one kind of constraints be translated to the other? How can correctness be proved for such a combined analysis?

For a typed language, perhaps ILP-based analyses can be phrased as type-based analyses [22]. Focusing on typable programs tends to make correctness proofs easier in the setting of set-based analysis for object-oriented and functional languages; perhaps also for ILP-based analyses?

4   Conclusion

Devices such as medical implants, smart cards, etc. lead to a demand for smaller and smaller computers so the issues of limited resources will continue to be with us. Writing the software can be done faster and more reliably in high-level languages, particularly with access to a resource-aware compiler. One of the main approaches to building resource awareness into a compiler is to use ILP-based static analysis. We have presented the formulation of four recent ILP-based static analyses in a uniform format for a simple example language. We have also outlined several open problems in ILP-based resource-aware compilation.

References

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

[2] Andrew Appel and Lal George. Optimal spilling for CISC machines with few registers. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of PLDI'01, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253, 2001.

[3] O. Avissar, R. Barua, 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*, November 2001.

[4] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. In *Proceedings of ICFP'00, ACM SIGPLAN International Conference on Functional Programming*, pages 209–220, 2000.

[5] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software – Practice & Experience*, 26(8):929–965, August 1996.

[6] Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of SAS'95, International Static Analysis Symposium*, pages 189–206. Springer-Verlag (*LNCS* 983), Glasgow, Scotland, September 1995.

[7] 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, November 30–December 2 1998.

[8] Ulrich Kremer. Optimal and near-optimal solutions for hard compilation problems. *Parallel Processing Letters*, 7(4), 1997.

[9] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 270–282. ACM Press, 2002.

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

[11] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In *Proceedings of Logics of Programs*, pages 219–224. Springer-Verlag (*LNCS* 193), 1985.

[12] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[14] R. Motwani, K. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report STAN-CS-TN-95-22, Department of Computer Science, Stanford University, 1995.

[15] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *LCTES'02, Languages, Compilers, and Tools for Embedded Systems joint with SCOPES'02, Software and Compilers for Embedded Systems*, pages 120–129, Berlin, Germany, June 2002.

[16] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[17] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE*, pages 357–373, April 1989.

[18] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[19] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe – definitely. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 161–170, San Diego, California, January 1998.

[20] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995. Preliminary version in Proceedings of CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (*LNCS* 787), pages 276–290, Edinburgh, Scotland, April 1994.

[21] Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.

[22] Jens Palsberg. Type-based analysis and applications. In *Proceedings of PASTE'01, ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools*, pages 20–27, Snowbird, Utah, June 2001. Invited paper.

[23] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

[24] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001. Preliminary version in Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 197–208, San Diego, California, January 1998.

[25] Jens Palsberg and Mitchell Wand. CPS transformation of flow information. *Journal of Functional Programming.* To appear.

[26] John 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.

[27] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *Proceedings of the 2002 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES '02)*, pages 2–11, June 2002.

[28] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, September 1989.

[29] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, October 1991.

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

[31] A. Stoutchinin. An integer linear programming model of software pipelining for the MIPS R8000 processor. In *PaCT'97, Parallel Computing Technologies, 4th International Conference*, pages 121–135. Springer-Verlag (*LNCS* 1277), 1997.

[32] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *Proceedings of TACS'94, Theoretical Aspects of Computing Software*, pages 224–243. Springer-Verlag (*LNCS* 789), 1994.

[33] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.

[34] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Proceedings of POPL'94, 21st Annual Symposium on Principles of Programming Languages*, pages 434–445, 1994.

[35] Mitchell Wand and Galen B. Williamson. A modular, extensible proof method for small-step flow analyses. In Daniel Le Métayer, editor, *Proceedings of ESOP 2002, 11th European Symposium on Programming, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April, 2002*, pages 213–227. Springer-Verlag (*LNCS* 2305), 2002.

[36] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *Proceedings of TAPSOFT'97, Theory and Practice of Software Development.* Springer-Verlag (*LNCS* 1214), 1997.

[37] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *Proceedings of PLDI '00, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 35(5):121–133, May 2000.

[38] H. P. Williams. *Model Building in Mathematical Programming*. Wiley & Sons, 1999.

[39] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Glossary

Compiler A software tool which translates a program in a high-level language to an equivalent program in a target language, typically machine code.

Data-flow equations A kind of set constraints of the form $X = Y$ where X and Y are set expressions. Data-flow equations are used in static analyses such as liveness analysis.

ILP Integer Linear Programming.

NP-complete The set of problems which are the hardest problems in the complexity class NP in that they are the ones most unlikely to be solvable in polynomial time. Examples of NP-complete problems include the Hamiltonian cycle and traveling salesman problems. At present, all known algorithms for NP-complete problems require time exponential in the problem size. It is unknown whether there are any faster algorithms. Therefore, NP-complete problems are usually solved using approximation algorithms, probabilistic methods, or heuristics.

Phase-ordering problem An optimizing compiler performs several optimization phases. Different orderings of the phases can yield different performance of the generated code. This is because one phase can benefit or suffer from being performed after another phase. The phase-ordering problem is to determine the best ordering of the optimization phases.

Set constraints Constraints of the form $X \subseteq Y$ where X and Y are set expressions. Such constraints are used in set-based static analyses.

Static analysis A technique that is used by a compiler to determine facts about a program that are useful for optimizations.

Type constraints Constraints of the form $X = Y$ or $X \leq Y$ where X and Y are type expressions. Such constraints are used in type-based static analyses.