Constraint-based Code Generation

Roberto Castañeda Lozano
SICS (Swedish Institute of Computer Science), Sweden
rcas@sics.se

Gabriel Hjort Blindell
KTH (School of ICT, KTH Royal Institute of Technology), Sweden
ghb@kth.se

Mats Carlsson
SICS, Sweden
matsc@sics.se

Frej Drejhammar
SICS, Sweden
frej@sics.se

Christian Schulte
KTH and SICS, Sweden
cschulte@sics.se

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, code generation, optimization*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*backtracking, scheduling*

General Terms

Languages, Design, Performance

Keywords

constraint programming, instruction selection, register allocation, instruction scheduling

1. INTRODUCTION

Compiler back-ends generate assembly code by solving three main tasks: instruction selection, register allocation and instruction scheduling. We introduce constraint models and solving techniques for these code generation tasks and describe how the models can be composed to generate code in unison. The use of constraint programming, a technique to model and solve combinatorial problems, makes code generation simple, flexible, robust and potentially optimal.

Today's back-ends typically generate code in stages: instruction selection is followed by register allocation and instruction scheduling. Each stage commonly executes a heuristic algorithm as taking optimal decisions is considered either too complex or computationally infeasible. Both staging and heuristics compromise the quality of the generated code and by design preclude optimal code generation. Furthermore, the use of heuristic algorithms complicates capturing common architectural features and adapting to new architectures and frequent processor revisions. This is particularly exacerbated for embedded processors which often impose additional challenges such as irregular register banks and very long instruction word (VLIW) capabilities. The use of a constraint model as opposed to staged and heuristic algorithms simplifies the construction of code generators and brings the potential of generating optimal code.

M-SCOPES '13, June 19-21, 2013, St. Goar, Germany.

Constraint-based code generation is organized as follows. A low-level intermediate representation (LIR) of a function in static single assignment form (SSA) and a processor description are taken as input. The different code generation tasks are translated into constraint models which capture the main challenges imposed by embedded processors: instruction selection on SSA graphs, global register allocation including multiple register banks, register packing, spilling and coalescing, and VLIW instruction scheduling [1]. The problems corresponding to each constraint model are finally solved in unison by a constraint solver as shown in Fig. 1.

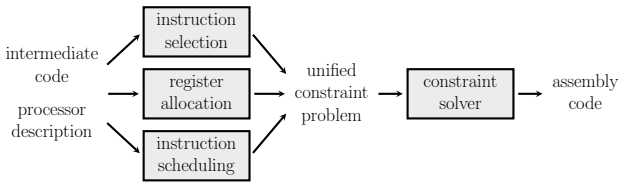

Figure 1: Code generation in unison.

2. CONSTRAINT PROGRAMMING

Constraint programming (CP) is a technique to model and solve combinatorial problems [5]. In CP, problems are modeled with decision variables and constraints that express relations among the variables. Constraint solvers interleave constraint propagation and search to find variable assignments that satisfy the constraints. Constraint propagation discards partial assignments that cannot take part in any solution. When no further propagation is possible, search decomposes the problem into several sub-problems in which the process is repeated. Constraint solvers can also solve optimization problems by applying branch-and-bound.

The main strength of CP is given by its ability to exploit global structure existing in many combinatorial problems. Such structure is captured by global constraints, which express relations among several variables. Global constraints give a two-fold benefit: they allow reusing recurrent patterns in modeling and can be exploited during propagation to avoid exploring parts of the search space which cannot lead to solutions. Modern constraint solvers provide an array of dedicated propagation algorithms for common global constraints.

3. INSTRUCTION SELECTION

The task of instruction selection is to transform the program from its intermediate form into target assembly instructions. In this approach, instruction selection is done by first converting the SSA graph into a directed acyclic graph by breaking the backward edges, and then performing pattern labeling via an augmented version of Hoffmann and O'Donnell's bottom-up algorithm [2]. Once all applicable pattern instances are detected, a constraint model is produced by forming constraints such that all nodes must be covered by exactly one pattern. The constraint model can be then composed with the register allocation and instruction scheduling models.

4. REGISTER ALLOCATION

The core task of register allocation is to assign program temporaries to either processor registers or memory, ensuring that temporaries that are live simultaneously are assigned to different registers.

Besides whole-function register assignment, the constraint model captures register packing (assign several small temporaries to the same register), handling of multiple register banks (subsuming spilling to memory) and coalescing (assign related temporaries to the same register to save move instructions). In particular, spilling, data transfer across register banks and coalescing are modeled in a novel, unified manner where different processor register banks and memory are represented in the same way [1].

The register allocation for a basic block is projected onto a rectangular area with registers in the horizontal dimension and clock cycles in the vertical dimension. Each temporary is represented as a rectangle with its definition and last use cycles as top and bottom coordinates. The horizontal coordinate is a decision variable that represents the register to which the temporary is assigned. A non-overlapping rectangles constraint, standard in CP, ensures that temporaries which are live simultaneously are assigned to different registers.

In a LIR temporaries have different sizes according to their source data types. Most processors allow several small temporaries to be packed into the same register. This feature, difficult to exploit for traditional approaches such as graph coloring, is naturally modeled with CP by giving each temporary rectangle a width proportional to its size [4]. Fig. 2 illustrates register packing for the Hexagon digital signal processor: 16-bit temporaries (t_1, t_2) have single width, 32-bit temporaries (t_3, t_4) have double width.

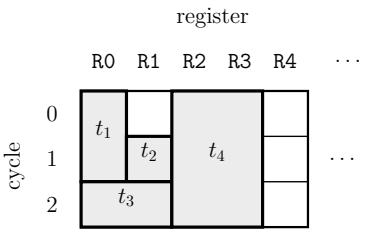

Figure 2: Register packing.

Register banks of limited capacity, inter-cluster communication in clustered VLIW processors and pre-assignments of temporaries to registers due to calling conventions require the generation of instructions that transfer data between register banks and memory. This is modeled uniformly for the first time by extending the input program with optional copy instructions. Copies can be turned into spill instructions (*load* and *store*), register-to-register transfer instructions (*move*) or be discarded. If a copy is discarded, its temporaries are coalesced by being assigned to the same register.

Register allocation is typically solved on whole functions (that is, *globally*) to reuse registers across basic blocks. To model global register allocation LSSA is introduced as a novel program form where SSA temporaries are split in each basic block. The relation to the original SSA temporaries is preserved by constraints that assign split temporaries to the same register.

5. INSTRUCTION SCHEDULING

The core task of instruction scheduling is to assign issue cycles to instructions such that the capacity of processor resources is never exceeded and data and control dependencies are satisfied. Scheduling becomes more challenging for VLIW processors which can issue several instructions every clock cycle.

A constraint model for VLIW instruction scheduling within basic blocks is available [1]. Similarly to the approach of Malik et al. [3], variables are introduced to represent the issue cycle of each instruction relative to the beginning of the basic block. Processor resource constraints ensure that the capacity of resources such as functional units is never exceeded. Constraint solvers provide special propagation algorithms for resource constraints as scheduling with resources is one of their most successful applications. Precedence constraints ensure that data and control dependencies are satisfied.

To exploit instruction-level parallelism in VLIW processors, instructions must be combined into valid bundles. Traditional code generators must be extended in non-trivial ways to support bundling. In contrast, the proposed CP scheduling model subsumes bundling as the number of instructions that can be issued in each cycle is not artificially constrained.

The close interdependencies between register allocation and instruction scheduling make it advantageous to solve both problems in unison. This is achieved by simply making the top and bottom coordinates of each temporary rectangle variable and relating them to the issue cycles of its definer and last user.

6. STATUS

The constraint models for register allocation and instruction scheduling are discussed in a conference article [1]. The initial results show that the quality of the generated code is on par with that of LLVM for a simple instruction set architecture such as MIPS32. This is expected since MIPS32 is designed to suit traditional code generation methods.

The current research focuses on implementing the instruction selection model, refining the register allocation model and generating code for embedded processors. The extra challenges imposed by these processors make them more likely to illustrate the advantages of the constraint-based approach.

7. REFERENCES

[1] R. Castañeda Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. Constraint-based register allocation and instruction scheduling. In *CP*, pages 750–766. Springer, 2012.

[2] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, Jan. 1982.

[3] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International Journal on Artificial Intelligence Tools*, 17(1):37–54, 2008.

[4] F. Pereira and J. Palsberg. Register allocation by puzzle solving. *SIGPLAN Not.*, 43:216–226, June 2008.

[5] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.