Constraint-based Register Allocation and Instruction Scheduling

Roberto Castañeda Lozano[1], Mats Carlsson[1], Frej Drejhammar[1], and Christian Schulte[2,1]

[1] SICS (Swedish Institute of Computer Science), Sweden
`{rcas,matsc,frej,cschulte}@sics.se`
[2] School of ICT, KTH Royal Institute of Technology, Sweden

Abstract. This paper introduces a constraint model and solving techniques for code generation in a compiler back-end. It contributes a new model for global register allocation that combines several advanced aspects: multiple register banks (subsuming spilling to memory), coalescing, and packing. The model is extended to include instruction scheduling and bundling. The paper introduces a decomposition scheme exploiting the underlying program structure and exhibiting robust behavior for functions with thousands of instructions. Evaluation shows that code quality is on par with LLVM, a state-of-the-art compiler infrastructure. The paper makes important contributions to the applicability of constraint programming as well as compiler construction: essential concepts are unified in a high-level model that can be solved by readily available modern solvers. This is a significant step towards basing code generation entirely on a high-level model and by this facilitates the construction of correct, simple, flexible, robust, and high-quality code generators.

1 Introduction

Compilers consist of a front-end and a back-end. The *front-end* analyzes the input program, performs architecture-independent optimizations, and generates an intermediate representation (IR) of the input program. The *back-end* takes the IR and generates assembly code for a particular processor. This paper introduces a constraint model and solving techniques for substantial parts of a compiler back-end and contributes an important step towards compiler back-ends that exclusively use a constraint model for code generation.

Today's back-ends typically generate code in stages: instruction selection (choose appropriate instructions for the program being compiled) is followed by register allocation (assign variables to registers or memory) and instruction scheduling (order instructions to improve their throughput). 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. Capturing common architectural features and adapting to new architectures and frequent processor revisions is difficult and error-prone with heuristic algorithms. The use of a constraint model as opposed to staged and heuristic algorithms facilitates the construction of correct, simple, flexible, and robust code generators with the potential to generate optimal code.

Approach. The paper uses LLVM [1] for the compiler front-end and assumes that instruction selection has already been done yielding a representation of the input program in SSA (static single assignment). The paper extends SSA by introducing LSSA (linear SSA), which represents programs as *basic blocks* (blocks of instructions without control flow, *blocks* for short) and relations of *temporaries* (program variables) among blocks.

A function in LSSA (the compilation unit in this paper) is used to generate a model for global register allocation (assign temporaries to registers for an entire function). The model combines several advanced aspects:

Multiple register banks also capture the spilling of temporaries to memory as just another register bank due to lack of available processor registers.
Temporary coalescing attempts to assign related temporaries to the same register in order to save move instructions.
Register packing can assign several small unrelated temporaries to the same register. For example, two 16-bit temporaries can be assigned to the upper and lower half of a 32-bit register.

Both multiple register banks and coalescing are modeled by optional copy instructions between temporaries identified as related in the LSSA representation. The model is extended to include instruction scheduling and instruction bundling (for executing several instructions in parallel). The single model accurately captures the interdependencies between register allocation and instruction scheduling. Hence, it faithfully reflects the trade-off between conflicting decisions during code generation.

The model is solved by decomposition, exploiting the underlying program structure as explicated in LSSA. First, the relation of temporaries among blocks is established followed by solving constraints for each block. The code generator exhibits robust behavior for functions with thousands of instructions, where we choose the `bzip2` program as part of the standard SPECint 2006 benchmark suite. Evaluation shows that code quality is on par with LLVM.

Key contributions. The paper makes the following contributions: – LSSA as a new program form explicating program structure used for modeling; – a constraint model unifying register allocation and instruction scheduling; – in particular, the register allocation model unifies multiple register banks, spilling, coalescing, and packing; and – a code generator based on a problem decomposition showing promising code quality and robustness.

Related approaches. Optimal register allocation and instruction scheduling have been approached with different optimization techniques, both in isolation and as an integrated problem.

Register allocation has been approached as an integer linear programming (ILP) problem [2, 3] and as a partitioned Boolean quadratic problem [4]. To the best of our knowledge, there has been no attempt to solve global register allocation with constraint programming (CP).

Instruction scheduling has typically been modeled as a resource-constrained scheduling problem. Local instruction scheduling has been approached with both CP [5, 6] and ILP [7, 8]. Proposed solutions for the global case include the use of CP [9], ILP [10], and special-purpose optimization algorithms [11, 12].

Different ILP and CP approaches have addressed the integration of both problems in a single model [13–16]. To the best of our knowledge, none of these approaches deals with all essential aspects of global register allocation such as coalescing and spilling.

The integration of instruction selection with these problems has also been considered, using both CP [17] and ILP [18, 19]. These approaches are limited to single blocks, and it is unclear how to extend them to handle entire functions robustly.

Plan of the Paper. Section 2 reviews SSA-based program representation whereas Sect. 3 introduces LSSA used for the constraint model in the paper. A constraint model for register allocation is introduced in Sect. 4 which is refined in Sect. 5 to integrate instruction scheduling. Section 6 discusses model limitations. Section 7 introduces a decomposition scheme which is evaluated in Sect. 8. Section 9 concludes the paper and presents future work.

2 SSA-based Program Representation

This section describes SSA (static single assignment) as a state-of-the-art representation for programs where processor instructions have already been selected. The factorial function, whose C source code is shown to the right, is used as a running example throughout the paper.

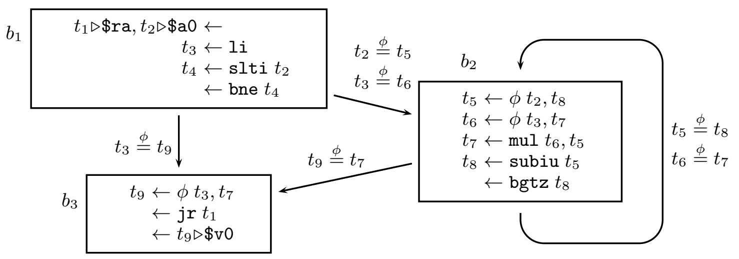
```
int factorial(int n) {
  int f = 1;
  while (n > 0) {
    f = f * n; n--;
  }
  return f;
}
```

A function is represented by its control-flow graph (CFG). The CFG's vertices correspond to blocks and its arcs define the control flow (jumps and branches) between blocks. A block consists of instructions and temporaries. Temporaries are storage locations corresponding to program variables after instruction selection. Other types of operands such as immediate values are not of interest in this context. An instruction is a triple represented as $D \leftarrow \texttt{op}\, U$, where D and U are the sets of *defined* and *used* temporaries and `op` is the *processor operation* that implements the instruction. For example, an instruction that uses a temporary t to define a temporary t' by executing the operation `neg` is represented as $t' \leftarrow \texttt{neg}\, t$. The remainder of the paper uses operations from MIPS32 [20], a simple instruction set chosen for ease of illustration.

A *program point* is located between two consecutive statements. A temporary is *live* at a program point if it holds a value that might be used by an instruction in the future. The *live range* of a temporary is the set of program points where it is live. Two temporaries *interfere* if their live ranges overlap. A temporary is *live-in* (respectively *live-out*) in a block if it is live at its entry (exit) point.

Architectural constraints and ABIs (application binary interfaces) predetermine the registers to which certain temporaries are assigned. A temporary t that is *pre-assigned* to a register `r` is represented by $t \triangleright$`r`.

SSA is a program form where temporaries are only defined once [21] (as is common, SSA in this paper means conventional SSA [22]). For programs where temporaries are redefined, SSA inserts *ϕ-functions* among the *natural instructions* derived from program statements and expressions. ϕ-functions disambiguate definitions of temporaries that depend on program control flow. For instance, in the factorial function, the return value might be defined by `int f = 1` or within the `while`-loop. In SSA, a ϕ-function is inserted defining a new return value that is equal to either `1` or to the the value of `f` computed in the loop. ϕ-functions define a congruence among temporaries, where two temporaries are *ϕ-congruent* if they are accessed by the same ϕ-function. Since ϕ-functions are not provided by processor instruction sets, their resolution is a prerequisite for generating executable code.



b_1
$t_1 \triangleright$`$ra`$, t_2 \triangleright$`$a0` $\leftarrow$
$t_3 \leftarrow$ `li`
$t_4 \leftarrow$ `slti` t_2
$\leftarrow$ `bne` t_4
$t_2 \overset{\phi}{=} t_5$
$t_3 \overset{\phi}{=} t_6$
b_2
$t_5 \leftarrow \phi\ t_2, t_8$
$t_6 \leftarrow \phi\ t_3, t_7$
$t_7 \leftarrow$ `mul` t_6, t_5
$t_8 \leftarrow$ `subiu` t_5
$\leftarrow$ `bgtz` t_8
$t_5 \overset{\phi}{=} t_8$
$t_6 \overset{\phi}{=} t_7$
$t_3 \overset{\phi}{=} t_9$
$t_9 \overset{\phi}{=} t_7$
b_3
$t_9 \leftarrow \phi\ t_3, t_7$
$\leftarrow$ `jr` t_1
$\leftarrow t_9 \triangleright$`$v0`

Fig. 1: MIPS32 instruction-selected function in SSA.

SSA simplifies the computation of liveness and interference. Since this simplification eases register allocation [23], SSA form is used as input to the code generator. Fig. 1 shows the control-flow graph of the running example transformed to SSA form with MIPS32 operations. Arc labels show the ϕ-congruence connecting temporaries related by ϕ-functions. Temporaries t_1, t_2 and t_9 are pre-assigned to the return address (`$ra`), first argument (`$a0`) and first return value (`$v0`) registers.

3 Program Representation for Register Allocation

This section introduces the program representation on which the model is based.

3.1 Register Allocation

Register allocation is the problem of assigning temporaries to either processor registers (hereafter called *registers*) or memory. Since access to registers is orders of magnitude faster than access to memory, it is desirable to assign all temporaries to registers. As the number of registers is limited, not all temporaries can be assigned a register. A first way to improve register utilization is to store temporaries only while they are live. This allows register allocation to assign several temporaries to the very same register provided the temporaries do not interfere with each other.

In general, even optimal register utilization does not guarantee the availability of processor registers for all temporaries. In this case, some temporaries must be stored in memory (that is, *spilled*). Since access to memory is costly, the decision of which temporaries are assigned to memory and at which program point they are assigned has high impact on the efficiency of the generated code.

The input program to register allocation may contain temporaries related by copy instructions (that is, instructions that copy the content of one temporary into another). Assigning these temporaries to the same register (that is, *coalescing* them) allows the removal of the corresponding copy instructions, improving the efficiency and size of the code.

Each temporary has a certain bit width (hereafter just called *width*) which is determined by the source data type that it represents. Many processors allow temporaries of different widths to be assigned to different parts of the same physical register. For example, Intel's x86 has 16-bit registers (`AX`) that combine pairs of 8-bit registers (`AH`, `AL`). For these processors, the ability to pack non-interfering temporaries into different parts of the same physical register is a key technique to improve register utilization [24].

Register allocation can be *local* or *global.* Local register allocation deals with one block at a time, spilling all temporaries that are live at block boundaries. Global register allocation yields better code by considering entire functions and can keep temporaries in the same register across blocks.

3.2 Linear Static Single Assignment Form

The live range of a temporary depends on where it is defined and used by instructions. In a model that captures simultaneous register allocation and instruction scheduling, the live ranges and their interferences are mutually dependent. Although SSA makes live range and interference computation easier than in a general program form, it is unclear how to model interference of temporaries with variable live ranges that can span block boundaries and follow branches.

To overcome this limitation and enable simple and direct modeling of live ranges, this paper introduces linear SSA (LSSA). LSSA is stricter than SSA in that each temporary is only defined and used within a single block. This property leads to simple, linear live ranges which do not span block boundaries and can be directly modeled as in Sect. 4. Furthermore, this simplification enables a problem decomposition that can be exploited for robust code generation, as Sect. 7 explains.

This paper is the first to take advantage of the explicit congruence structure that LSSA defines even though this structure appears in some SSA construction approaches [25, 26].

To restrict live ranges to single blocks, SSA ϕ-functions are generalized to *delimiter instructions* (hereafter just called *delimiters*). These instructions are placed at the block boundaries and are not part of the generated code. Each block contains two delimiters: an *in-delimiter* which defines the live-in temporaries and an *out-delimiter* which uses the live-out temporaries.

In LSSA, the live range of a temporary cannot span the boundaries of the block where it is defined. Liveness across blocks in the original program is captured by a new definition of temporary congruence, relating temporaries that represent the same original temporary in different blocks. This congruence generalizes the ϕ-congruence defined for SSA.

Fig. 2 shows the control-flow graph of the running example transformed to LSSA. Arc labels show the generalized congruence. The temporary t_1 which is live across all branches in Fig. 1 corresponds to the congruent temporaries t_1, t_5 and t_{10} in Fig. 2, each of them having a linear live range. The figure shows that the only link between blocks in LSSA is given by temporary congruences.

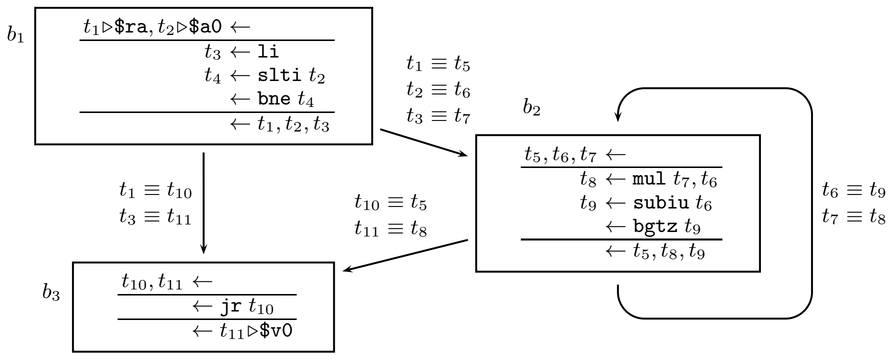


Fig. 2: MIPS32 instruction-selected function in LSSA.

LSSA can be easily constructed from SSA by applying the following steps to each block:

1. introduce delimiters;
2. for each live-in temporary, introduce a new definition by the in-delimiter; for each live-out temporary, introduce a new use by the out-delimiter (applying the liveness definition by Sreedhar et al. [22]);
3. remove all ϕ-functions;
4. connect the new definitions and uses with their corresponding live-in and live-out temporaries by defining congruences.

3.3 Copies

Spilling requires copying the contents of temporaries to new temporaries that can then be assigned to different processor registers or to memory. This is captured in the model by extending the program representation with the *copy* instruction type, similarly to Appel and George's approach [3]. A copy replicates the content of a temporary t_s to a new temporary t_d. To allow t_s and t_d to be assigned to different types of locations such as registers or memory, the copy is implemented by the execution of one of a set of alternative operations $\{o_1, o_2, \ldots, o_n\}$ and represented as $t_d \leftarrow \{o_1, o_2, \ldots, o_n\}\ t_s$.

The way in which a program is extended with copies depends on the processor. In load/store processors such as MIPS32, arithmetic/logic operations define and use temporaries in registers. In this setting, register allocation needs to be able to copy a temporary defined in a register to another register or to memory. If a temporary has been copied to memory (that is, spilled), it must be copied back to a register before its use by an arithmetic/logic operation. In MIPS32, the program is extended with copies of the form $t_d \leftarrow \{\texttt{move}, \texttt{sw}\}\ t_s$ after the definition of t_s in a register and $t_d \leftarrow \{\texttt{move}, \texttt{lw}\}\ t_s$ before the use of t_d in a register, where the operations `move`, `sw` and `lw` implement register-to-register, register-to-memory and memory-to-register copies. Fig. 3 shows how the function in Fig. 2 is extended with such copies.

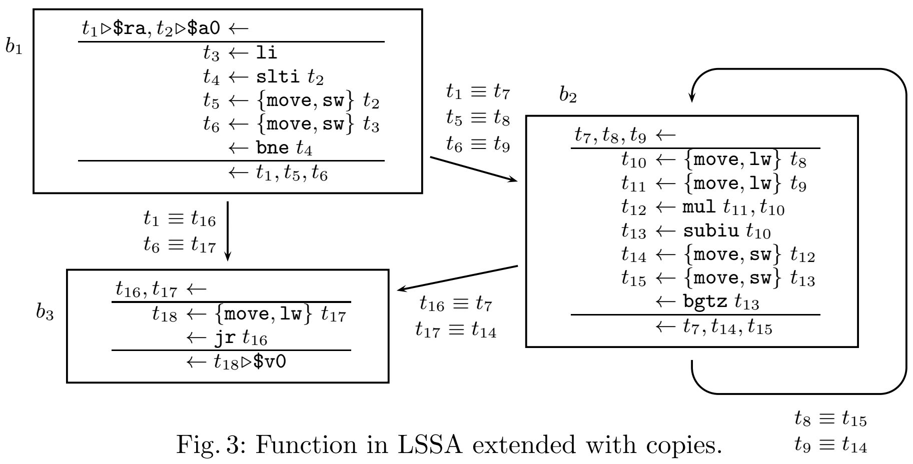

b_1
$t_1 \triangleright \texttt{\$ra}, t_2 \triangleright \texttt{\$a0} \leftarrow$
$t_3 \leftarrow \texttt{li}$
$t_4 \leftarrow \texttt{slti}\ t_2$
$t_5 \leftarrow \{\texttt{move}, \texttt{sw}\}\ t_2$
$t_6 \leftarrow \{\texttt{move}, \texttt{sw}\}\ t_3$
$\leftarrow \texttt{bne}\ t_4$
$\leftarrow t_1, t_5, t_6$
$t_1 \equiv t_7$
$t_5 \equiv t_8$
$t_6 \equiv t_9$
b_2
$t_7, t_8, t_9 \leftarrow$
$t_{10} \leftarrow \{\texttt{move}, \texttt{lw}\}\ t_8$
$t_{11} \leftarrow \{\texttt{move}, \texttt{lw}\}\ t_9$
$t_{12} \leftarrow \texttt{mul}\ t_{11}, t_{10}$
$t_{13} \leftarrow \texttt{subiu}\ t_{10}$
$t_{14} \leftarrow \{\texttt{move}, \texttt{sw}\}\ t_{12}$
$t_{15} \leftarrow \{\texttt{move}, \texttt{sw}\}\ t_{13}$
$\leftarrow \texttt{bgtz}\ t_{13}$
$\leftarrow t_7, t_{14}, t_{15}$
$t_1 \equiv t_{16}$
$t_6 \equiv t_{17}$
b_3
$t_{16}, t_{17} \leftarrow$
$t_{18} \leftarrow \{\texttt{move}, \texttt{lw}\}\ t_{17}$
$\leftarrow \texttt{jr}\ t_{16}$
$\leftarrow t_{18} \triangleright \texttt{\$v0}$
$t_{16} \equiv t_7$
$t_{17} \equiv t_{14}$
$t_8 \equiv t_{15}$
$t_9 \equiv t_{14}$

Fig. 3: Function in LSSA extended with copies.

4 Register Allocation and Packing

This section describes the constraint model for register allocation and packing. The constraint model is parameterized with respect to a program in LSSA and a processor. The entire model, extended with instruction scheduling, is shown in Fig. 4. The main text contains references to the formulas (1-11) from the figure. Sections 4.1 and 4.2 discuss local and global register allocation. Section 4.3 refines the model to handle register packing.

Program parameters

B, I, T	sets of blocks, instructions and temporaries
$\text{ins}(b)$	set of instructions of block b
$\text{tmp}(b)$	set of temporaries defined and used within block b
$\text{tmp}(i)$	set of temporaries defined and used by instruction i
$\text{definer}(t)$	instruction that defines temporary t
$\text{users}(t)$	set of instructions that use temporary t
$t \equiv t'$	whether temporaries t and t' are congruent
$\text{dep}(b)$	dependency graph of the instructions of block b
$\text{ops}(i)$	set of alternative operations implementing i (singleton for non-copies)
$\text{width}(t)$	number of register atoms that temporary t occupies
$t \triangleright \mathtt{r}$	whether temporary t is pre-assigned to register `r`
$\text{src}(i)$	source temporary of copy instruction i
$\text{dst}(i)$	destination temporary of copy instruction i
$\text{freq}(b)$	estimation of the execution frequency of block b

Processor parameters

$\text{space}(i, \mathtt{op}, t)$	register space in which instruction i implemented by `op` accesses t
$\text{forbidden}(t)$	forbidden first assigned register atoms for temporary t
$\text{lat}(\mathtt{op})$	latency of operation `op`
R	set of processor resources
$\text{cap}(r)$	capacity of processor resource r
$\text{use}(\mathtt{op}, r)$	units of processor resource r used by operation `op`
$\text{dur}(\mathtt{op}, r)$	duration of usage of processor resource r by operation `op`

Variables

$r_t \in \mathbb{N}_0$	register to which temporary t is assigned
$o_i \in \mathbb{N}_0$	operation that implements instruction i
$c_i \in \mathbb{N}_0$	issue cycle of instruction i relative to the beginning of its block
$a_i \in \{0, 1\}$	whether instruction i is active
$ls_t \in \mathbb{N}_0$	start of live range of temporary t
$le_t \in \mathbb{N}_0$	end of live range of temporary t

Register allocation constraints

$$ls_t = c_{\text{definer}(t)} \quad \forall t \in T; \qquad le_t = \max_{u \in \text{users}(t)} c_u \quad \forall t \in T \tag{1}$$

$$a_i \quad \forall i : (i \text{ is a natural instruction}) \vee (i \text{ is a delimiter}) \tag{2}$$

$$o_i \in \text{ops}(i) \cup \{\mathtt{null}\} \quad \forall i : i \text{ is a copy} \tag{3}$$

$$\text{disjoint2}(\{\langle r_t, r_t + \text{width}(t), ls_t, le_t \rangle : t \in \text{tmp}(b)\}) \quad \forall b \in B \tag{4}$$

$$o_i = \mathtt{op} \implies r_t \in \text{space}(i, \mathtt{op}, t) \quad \forall t \in \text{tmp}(i), \forall \mathtt{op} \in \text{ops}(i), \forall i \in I \tag{5}$$

$$r_{\text{src}(i)} \neq r_{\text{dst}(i)} \iff a_i \quad \forall i : i \text{ is a copy} \tag{6}$$

$$r_t = \mathtt{r} \quad \forall t \in T : t \triangleright \mathtt{r} \tag{7}$$

$$r_t = r_{t'} \quad \forall t, t' \in T : t \equiv t' \tag{8}$$

$$r_t \notin \text{forbidden}(t) \quad \forall t \in T \tag{9}$$

Instruction scheduling constraints

$$a_i \wedge a_j \implies c_j \geq c_i + \text{lat}(o_i) \quad \forall (i, j) \in \text{edges}(\text{dep}(b)), \forall b \in B \tag{10}$$

$$\text{cumulative}(\{\langle c_i, \text{dur}(o_i, r), a_i \times \text{use}(o_i, r) \rangle : i \in \text{ins}(b)\}, \text{cap}(r)) \quad \forall b \in B, \forall r \in R \tag{11}$$

Fig. 4: Model parameters, variables, and constraints.

4.1 Local Register Allocation

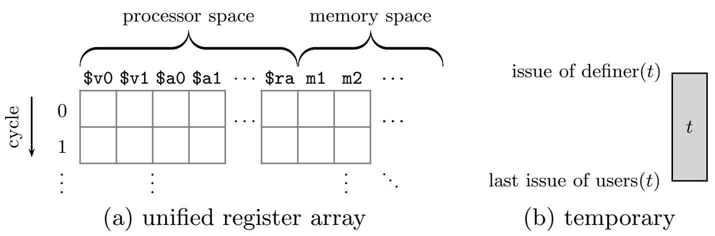
The variables in the constraint model are described in Fig. 4. A valid assignment of the register (r_t) and operation (o_i) variables constitutes a solution to the register allocation problem. When solving this problem in isolation, the issue cycle (c_i) and live range (ls_t, le_t) variables (1) are pre-assigned and can be seen as program parameters. They act as variables when the model is extended with instruction scheduling as described in Sect. 5.

Natural instructions and delimiters define the meaning of a program and must be active (2). Unlike natural instructions and delimiters, a copy i might be implemented by different operations or be inactive. To support the latter case, the domain of its variable o_i is extended with a virtual operation `null` (3). Delimiters are implemented by virtual `in` and `out` operations. A solution to the register allocation problem implies deciding whether a copy i is active (a_i) and which operation implements it (o_i).

A processor typically contains one or more register banks which can be directly accessed by instructions. Traditional register allocation treats memory and different register banks as separate entities, which leads to specialized methods for different aspects of register allocation such as spilling and dealing with multiple register banks. We consider a unified register array that fully integrates these aspects. A unified register array is a sequence of register *spaces*. A space is a sequence of related registers. Spaces capture different processor register banks as well as *memory registers* (representing memory locations on the runtime stack). A memory space contains a practically infinite sequence of memory registers (`m1`, `m2`, ...). Fig. 5a shows the unified register array for MIPS32.

To the best of our knowledge, this paper presents the first application of a unified register array to integrate different aspects of register allocation in the context of native code generation.

Local register allocations can be projected onto a rectangular area. The horizontal dimension represents the registers in the unified register array, whereas the vertical dimension represents time in clock-cycles (see Fig. 5a). In this projection, each temporary t is represented as a rectangle with width(t) = 1. The top and bottom coordinates of the rectangle reflect the issue cycle of its *definer* and the last issue cycle of its *users*. The horizontal coordinate represents the register to which the temporary is assigned (see Fig. 5b).



(a) unified register array (b) temporary

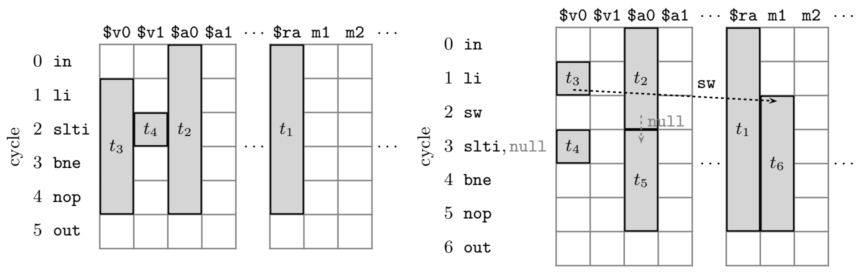
Fig. 5: Geometric interpretation of local register allocation.

In this representation, two temporaries interfere when their rectangles overlap vertically. The non-overlapping rectangles constraint disjoint2 [27] enforces interfering temporaries to be assigned to different registers (4). Fig. 6a shows a register allocation for a given instruction schedule of block b_1 from Fig. 2. In this schedule, all temporaries interfere with each other and must be assigned to different processor registers.

Due to architectural constraints, operations can only access their operands in certain spaces. The operation that implements an instruction determines the space to which its temporaries are allocated (5).

A copy i from a temporary $\mathrm{src}(i)$ to a temporary $\mathrm{dst}(i)$ is active when these temporaries are not coalesced (6). Fig. 6b shows a register allocation for the block b_1 in Fig. 3, where dotted arrows connect copy-related temporaries. t_2 and t_5 are coalesced, rendering their copy inactive. The copy from t_3 to t_6 is implemented by `sw` and represents a spill to `m1`. This frees `$v0` from cycle 2 onwards and allows t_4 to reuse it, reducing the set of used processor registers from $\{\texttt{\$v0}, \texttt{\$v1}, \texttt{\$a0}, \texttt{\$ra}\}$ in Fig. 6a to $\{\texttt{\$v0}, \texttt{\$a0}, \texttt{\$ra}\}$ in Fig. 6b.

Some temporaries are pre-assigned to registers (7).



(a) without copies (b) spilling t_3 enables reuse of `$v0`

Fig. 6: Two register allocations for block b_1.

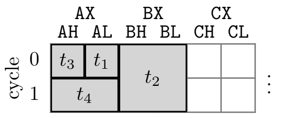
4.2 Global Register Allocation

In LSSA, the global relation between temporaries is solely captured by temporary congruences, which leads to a direct extension of the local problem. Temporaries whose live ranges span block boundaries are decomposed in LSSA into congruent temporaries with linear live ranges. Congruent temporaries resulting from this decomposition represent the same original temporary and are assigned to the same register (8).

4.3 Register Packing

The register allocation model described in this section is extended with register packing following the approach of Pereira and Palsberg [28]. Registers are decomposed into register *atoms*. An atom is the minimum part of a physical register that can be referenced by an operation (for example, `AH` in x86). A space is a sequence of atoms, each of which corresponds to a column in the unified register array. width(t) becomes a program parameter giving the number of atoms that the temporary t occupies. The variable r_t represents the first of the atoms to which t is assigned. Enforcing non-interference among temporaries assigned to the same register (4) thus becomes isomorphic to rectangle packing.

Most processors restrict the combinations of atoms out of which wider registers can be formed. For example, double-width temporaries such as t_2 and t_4 in the figure to the right cannot be assigned to {AL, BH} or to {BL, CH} in the 16-bit x86 register array. Such forbidden atoms are removed from the register variable domains (9).

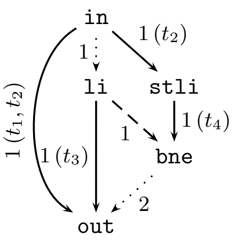


5 Instruction Scheduling and Bundling

This section describes how to extend the register allocation model with instruction scheduling and bundling.

Local instruction scheduling. To model instruction scheduling, the issue cycle c_i of each instruction i and the (derived) live range (ls_t, le_t) of each temporary t become variables (1).

Flow of data and control cause dependencies among instructions. The dependencies in a block b form a *dependency graph* dep(b) with instructions as vertices. For example, the figure to the right shows dep(b_1) from Fig. 2. Solid, dashed and dotted arcs respectively represent data, control and artificial dependencies. The latter are added to make delimiters first and last in any topological ordering. These dependencies dictate precedence constraints among instructions. The minimum issue distance in a precedence (i, j) is equal to the latency of the parent lat(o_i). Precedence constraints are only effective when both instructions are active (10).



Operations share limited processor resources such as functional units and data buses. This is naturally captured as a task-resource model with a cumulative constraint [29] for each processor resource and block. These constraints include a task for each active instruction in the block (11).

Instruction bundling. Very Long Instruction Word and Explicitly Parallel Instruction Computing processors can issue several instructions every clock cycle. To exploit this capability, instructions must be combined into valid bundles satisfying precedence (10) and processor resource (11) constraints [30]. The presented scheduling model already subsumes bundling, by interpreting sets of instructions issued in the same cycle as bundles and giving the appropriate processor resource configuration.

6 Model Limitations

This section discusses some model limitations to be addressed in the future.

$$\begin{array}{l} \ldots \\ t_j \leftarrow t_i \\ \ldots \\ \leftarrow \ldots, t_i, t_j, \ldots \end{array}$$

Limited coalescing. The constraint model employs a simple definition of interference with a direct geometrical interpretation: two temporaries *interfere* when the rectangles representing their live ranges overlap. Copy-related temporaries that do not interfere can often be coalesced, rendering their copy instructions inactive and saving execution cycles. That is the case, for example, for t_2 and t_5 in Fig. 6b. Relaxed definitions of interference have been proposed which expose more coalescing opportunities [31]. In the example to the right, t_i and t_j can be coalesced into a single temporary since both hold the same value. However, in the constraint model this makes their rectangles overlap, which is not allowed. This limitation, shared by the related register allocation approaches mentioned in the introduction, is significantly mitigated in the constraint model by the possibility of rearranging live ranges through instruction scheduling.

$$\begin{array}{l} \ldots \\ t_2 \leftarrow \{\texttt{move}, \texttt{sw}\}\ t_1 \\ \ldots \\ t_3 \leftarrow \{\texttt{move}, \texttt{lw}\}\ t_2 \\ \ldots \leftarrow \texttt{op}\ t_3 \\ \ldots \\ t_4 \leftarrow \{\texttt{move}, \texttt{lw}\}\ t_2 \\ \ldots \leftarrow \texttt{op}\ t_4 \\ \ldots \end{array}$$

Spilling reused temporaries. Once a temporary is spilled, it must be loaded into a register before every use. If the temporary is used multiply, it might be desirable to load it to a register once and keep it there for the remaining uses. The example to the right illustrates this limitation: if t_1 is spilled to memory by a `sw` operation, t_2 must be loaded into a register twice by `lw` operations, once for each operation `op`. Fortunately, in SSA most of the temporaries are only used once [32], and this percentage is even larger in LSSA since SSA temporaries are further decomposed.

7 Decomposition-based Code Generation

This section introduces a decomposition scheme and a code generator that exploit the properties of LSSA.

The main property of LSSA is that temporaries are live in single blocks only. All temporaries accessed by delimiters (*global* temporaries) are pre-assigned or congruent to temporaries in other blocks. For example, the global temporaries in Fig. 3 are $\{t_1, t_2, t_5, t_6, t_7, t_8, t_9, t_{14}, t_{15}, t_{16}, t_{17}, t_{18}\}$. The only link between different blocks in the model is given by the congruence constraints (8), which relate the register variables r_t of global temporaries. Thus, once these variables are assigned, the rest of the register allocation and instruction scheduling problem can be solved independently for each block. For example, assuming that $\{t_2, t_5\}$ and $\{t_3, t_6\}$ have been coalesced, the problem variables that remain to be assigned for block b_1 in Fig. 3 are the register $\{r_{t_3}, r_{t_4}\}$ and issue cycle variables.

Based on this observation, we devise a decomposition scheme that significantly reduces the search space. It proceeds by first solving the global problem (assigning the r_t variables of global temporaries) and then solving a local problem for each block (assigning the remaining variables).

The objective is to minimize execution cycles according to an estimate of block execution frequency:

$$\text{minimize} \sum_{b \in B} \text{freq}(b) \times \max_{i \in \text{ins}(b)} c_i \tag{12}$$

Fig. 7 depicts the architecture of the code generator. SSA functions are transformed to LSSA and extended with copies as described in Sect. 3. Then, a satisfaction problem with all constraints is solved, assigning global temporaries to registers. This assignment constitutes a *global solution*. As variable selection heuristic, the global solver branches first on the largest temporary congruence class ($\{t_6, t_9, t_{14}, t_{17}\}$ in Fig. 3). As value selection heuristic, it performs a cost-benefit analysis to determine the most effective register for each temporary. The benefit component estimates the saved spilling overhead, while the cost component is based on an estimate of the increased space occupation. This analysis is parameterized with an aggressiveness factor to direct the heuristic towards either the benefit or the cost component. In the example, all temporaries are assigned to processor registers regardless of this parameter, since spilling is costly and the processor register space occupation is low.

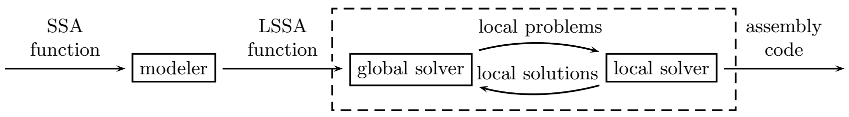


Fig. 7: Architecture of the code generator.

Once a global solution is found, an optimization problem with all but the congruence constraints (8) is solved for each block, seeking a locally optimal assignment of the remaining variables. This assignment constitutes a *local solution*. The search starts by branching on the copy activation variables a_i, trying first to inactivate the copy. Then, the solver branches on the c_i variables in topological dependency order, selecting the earliest cycle first. Finally, local temporaries are assigned to registers by assigning their r_t variables. The global and local solutions are then combined and the total cost is computed. This process is repeated, increasing the aggressiveness of the global solver in each iteration until it proves optimality or reaches a time limit.

8 Evaluation

Two essential characteristics of the code generator are examined by the experiments: the quality of the generated code and its solving time. The global and local solvers are implemented with the constraint programming system Gecode 3.7.3 [33]. As input, we have used functions from the C program `bzip2` as a representative of the standard SPECint 2006 benchmark suite, optimized the intermediate code of each function and selected MIPS32 instructions using the LLVM 3.0 compiler infrastructure. The instruction-selected functions are passed as input to the code generator for this experiment. The global and local solvers are run with a time limit of 10 and 3 seconds respectively. All experiments are run using sequential search on a Linux machine with a quad-core Intel Core i5-750 processor and 4 GB of main memory.

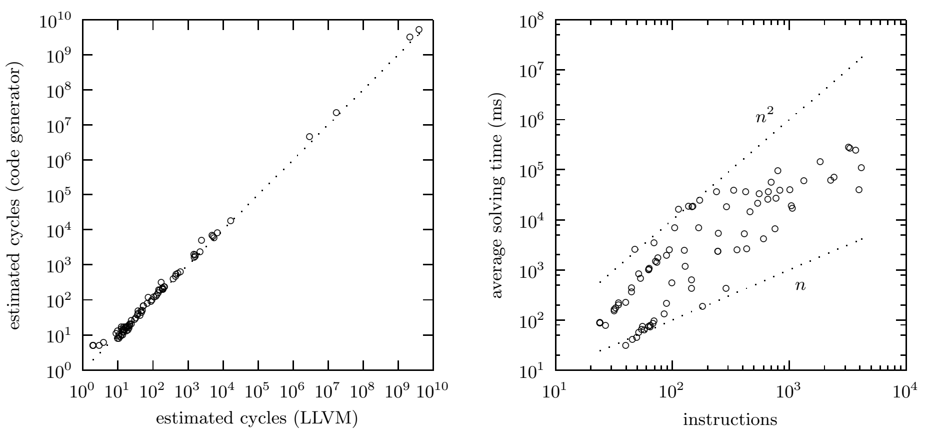
Code quality. Lack of post-code-generation support (including generation of assembly directives) prevents running the generated code. Therefore, to measure its quality, we estimate the number of execution cycles by computing the value of the objective function (12). This measure is computed for the code generated by both our system and LLVM's register allocator (based on priority-based coloring [34]) and instruction scheduler (based on list-scheduling [35]). These LLVM components are run with the following flags: `-O3 -enable-mips-delay-filler -disable-post-ra -disable-tail-duplicate -disable-branch-fold`. This comparison is meaningful since a) the input to the LLVM components is the same as to our system, and b) the optimization flags given to LLVM are aligned with the objective function (12). For example, `-O3` emphasizes speed at the possible expense of code size. Furthermore, the same optimization level is enforced after code generation by disabling *tail duplication* and *branch folding* in LLVM.

From a total of 106 functions in `bzip2`, the code generator pre-processes and solves 86 functions. The remaining ones cannot yet be pre-processed by the `modeler` module (see Fig. 7) because of lack of support for the MIPS32 floating-point extension and incompleteness of the interface to LLVM's instruction selector. Fig. 8a shows the estimated execution cycles of each solved function. The figure shows that the code generator is competitive with LLVM, a state-of-the-art compiler infrastructure, in terms of code quality. The cases in which LLVM generates better code are due to a) few of the 86 functions being solved optimally because of global and local time-outs, and b) the limitations discussed in Sect. 6, which in particular prevent coalescing certain copies in blocks that belong to deeply nested loops.

Solving time. The global and local solvers dominate the execution time of the code generator. Fig. 8b shows, for each function in the first experiment, the average time to solve the constraint problems and the number of instructions. The average is calculated on 10 iterations, where the maximum coefficient of variation per function is 10%. The figure reveals a sub-quadratic relation between solving time and size of the compiled functions, confirming the robustness of the code generator for functions with thousands of instructions. This robust behavior is due partially to the effect of solver time-outs on all non-trivial functions.

9 Conclusion and Future Work

This paper introduces a constraint model capturing global register allocation and local instruction scheduling as two main tasks of code generation. In particular, the model of register allocation combines all essential aspects in this problem: generalized spilling, coalescing and register packing. The paper introduces LSSA to enable a direct model of global register allocation and a problem decomposition. A code generator is presented that exploits this decomposition to achieve robust behavior. Experiments show that it generates code that is competitive with a state-of-the-art compiler infrastructure for functions of thousands of instructions.



(a) code quality vs. LLVM (b) solving time vs. instructions

Fig. 8: Evaluation of the decomposition-based code generator.

Future work. This paper presents an important step towards basing code generation entirely on a high-level model as opposed to limited heuristic algorithms. There is considerable future work in this direction. A first step is to address the limitations from Sect. 6. Also, although the code generator shows robust behavior, there is still a significant efficiency gap with respect to state-of-the-art compiler infrastructure such as LLVM. We have identified several opportunities to improve the efficiency of the code generator by applying standard modeling techniques: breaking symmetries in the dependency graph [6] and in the register array, strengthening the geometric reasoning on the register array by coalescing-aware custom propagators and inferring implied constraints to improve propagation are some examples. Another possible improvement is to refine the decomposition described in Sect. 7 into a Benders-like scheme [36], where the local solver feeds back problem knowledge to the global solver.

We plan to study how to extend the model from this paper with instruction selection and other code generation problems to further improve code quality. Finally, we intend to evaluate the generality and flexibility of the model by targeting more challenging architectures such as digital signal processors and Intel's x86. We conjecture that the additional difficulties imposed by these architectures will only but highlight the advantages of a constraint-based approach.

Acknowledgments

The research reported in this paper has been partially funded by LM Ericsson AB and the Swedish Research Council (VR 621-2011-6229). The authors are grateful for helpful comments from Gabriel Hjort Blindell and the anonymous reviewers.

References

1. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. (March 2004)
2. Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0-1 integer programming. Software – Practice and Experience **26** (August 1996) 929–965
3. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. SIGPLAN Not. **36** (May 2001) 243–253
4. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. SIGPLAN Not. **37** (June 2002) 139–148
5. Ertl, M., Krall, A.: Optimal instruction scheduling using constraint logic programming. In: PLILP. Volume 528 of LNCS., Springer (1991) 75–86
6. Malik, A.M., McInnes, J., van Beek, P.: Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. International Journal on Artificial Intelligence Tools **17**(1) (2008) 37–54
7. Leupers, R., Marwedel, P.: Time-constrained code compaction for DSP's. IEEE Transactions on Very Large Scale Integration Systems **5** (March 1997) 112–122
8. Wilken, K., Liu, J., Heffernan, M.: Optimal instruction scheduling using integer programming. SIGPLAN Not. **35** (May 2000) 121–133
9. Malik, A.M., Chase, M., Russell, T., van Beek, P.: An application of constraint programming to superblock instruction scheduling. In: CP. Volume 5202 of LNCS., Springer (2008) 97–111
10. Winkel, S.: Exploring the performance potential of Itanium processors with ILP-based scheduling. In: CGO, IEEE (2004) 189–200
11. Chou, H.C., Chung, C.P.: An optimal instruction scheduler for superscalar processor. IEEE Transactions on Parallel and Distributed Systems **6** (March 1995) 303–313
12. Shobaki, G., Wilken, K.: Optimal superblock scheduling using enumeration. In: MICRO, IEEE (2004) 283–293
13. Gebotys, C.H., Elmasry, M.I.: Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In: DAC, ACM (1991) 2–7
14. Kästner, D.: PROPAN: A retargetable system for postpass optimisations and analyses. In: LCTES, Springer (2001) 63–80
15. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Trans. Des. Autom. Electron. Syst. **8** (July 2003) 355–383
16. Nagarakatte, S.G., Govindarajan, R.: Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In: CC. Volume 4420 of LNCS., Springer (2007) 126–140
17. Bashford, S., Leupers, R.: Phase-coupled mapping of data flow graphs to irregular data paths. Design Automation for Embedded Systems (1999) 119–165
18. Wilson, T., Grewal, G., Halley, B., Banerji, D.: An integrated approach to retargetable code generation. In: ISSS, IEEE (1994) 70–75
19. Eriksson, M.V., Skoog, O., Kessler, C.W.: Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In: SCOPES. (2008) 11–20
20. Sweetman, D.: See MIPS Run, Second Edition. Morgan Kaufmann (2006)
21. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM TOPLAS **13**(4) (October 1991) 451–490

22. Sreedhar, V., Ju, R., Gillies, D., Santhanam, V.: Translating out of static single assignment form. In: SAS. Volume 1694 of LNCS., Springer (1999) 849–849
23. Hack, S., Grund, D., Goos, G.: Register allocation for programs in SSA-form. In: CC. Volume 3923 of LNCS., Springer (2006) 247–262
24. Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. SIGPLAN Not. **39** (June 2004) 277–288
25. Appel, A.W.: SSA is functional programming. SIGPLAN Not. **33**(4) (April 1998) 17–20
26. Aycock, J., Horspool, R.N.: Simple generation of static single-assignment form. In: CC. Volume 1781 of LNCS., Springer (2000) 110–124
27. Beldiceanu, N., Carlsson, M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In: CP. Volume 2239 of LNCS., Springer (2001) 377–391
28. Pereira, F., Palsberg, J.: Register allocation by puzzle solving. SIGPLAN Not. **43** (June 2008) 216–226
29. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling **17**(7) (1993) 57–73
30. Kessler, C.W.: Compiling for VLIW DSPs. In: Handbook of Signal Processing Systems. Springer (2010) 603–638
31. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages **6**(1) (1981) 47–57
32. Boissinot, B., Hack, S., Grund, D., Dupont de Dinechin, B., Rastello, F.: Fast liveness checking for SSA-form programs. In: CGO, ACM (2008) 35–44
33. Gecode Team: Gecode: generic constraint development environment. www.gecode.org (2006)
34. Chow, F., Hennessy, J.: Register allocation by priority-based coloring. SIGPLAN Not. **19**(6) (June 1984) 222–232
35. Rau, B.R., Fisher, J.A.: Instruction-level parallel processing: history, overview, and perspective. J. Supercomput. **7** (May 1993) 9–50
36. Hooker, J.N., Ottosson, G.: Logic-based Benders decomposition. Mathematical Programming **96**(1) (2003) 33–60