Combinatorial Spill Code Optimization and Ultimate Coalescing

Roberto Castañeda Lozano Mats Carlsson
SCALE, Swedish Institute of Computer Science, Sweden
{rcas,matsc}@sics.se

Gabriel Hjort Blindell Christian Schulte
SCALE, School of ICT, KTH Royal Institute of Technology, Sweden
{ghb,cschulte}@kth.se

Abstract

This paper presents a novel combinatorial model that integrates global register allocation based on ultimate coalescing, spill code optimization, register packing, and multiple register banks with instruction scheduling (including VLIW). The model exploits alternative temporaries that hold the same value as a new concept for ultimate coalescing and spill code optimization.

The paper presents Unison as a code generator based on the model and advanced solving techniques using constraint programming. Thorough experiments using MediaBench and a processor (Hexagon) that are typical for embedded systems demonstrate that Unison: is robust and scalable; generates faster code than LLVM (up to 41% with a mean improvement of 7%); possibly generates optimal code (for 29% of the experiments); effortlessly supports different optimization criteria (code size on par with LLVM).

Unison is significant as it addresses the same aspects as traditional code generation algorithms, yet is based on a simple integrated model and robustly can generate optimal code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, code generation, optimization; D.3.2 [*Programming Languages*]: Language Classifications—constraint and logic languages; I.2.8 [*Artificial Intelligence*]: Problem Solving, Control Methods, and Search—backtracking, scheduling

Keywords spill code optimization; ultimate coalescing; combinatorial optimization; register allocation; instruction scheduling

1. Introduction

Register allocation and instruction scheduling are essential aspects of generating assembly code during compilation. They are particularly relevant for embedded processors such as Qualcomm's *Hexagon* or Recore Systems' *Xentium* with additional challenges such as very long instruction word (VLIW) capabilities and irregular register banks. This paper presents a novel combinatorial model and Unison as a code generator using the model. The model is formally expressed by variables and relations (constraints) between variables. Unison uses constraint programming as a combinatorial optimization technique to solve the model and thereby generates potentially optimal assembly code for a given input function and processor architecture. Unison addresses all major subproblems of integrated register allocation and instruction scheduling. This approach overcomes significant limitations of previous work while being scalable and robust (wrt. different input functions) and produces better assembly code than traditional algorithms.

Today's compilers typically generate assembly 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 to be computationally infeasible. By design, both staging and heuristic algorithms compromise the quality of the generated code. Moreover, heuristic algorithms are difficult to adapt to new architectural features and frequent processor revisions, particularly for embedded processors. Using instead a combinatorial model simplifies the construction of compilers while generating potentially optimal code.

Existing combinatorial models of register allocation and instruction scheduling predefine which instructions access each *temporary* (program variable) and thus do not support the substitution of temporaries that hold the same value. This is a significant limitation that precludes two essential optimizations: *spill code optimization* (remove unnecessary memory access instructions inserted during register allocation) and *ultimate coalescing* (remove unnecessary register-to-register copy instructions considering the value of each temporary). This paper introduces *alternative temporaries* as an approach that supports the substitution of temporaries and thus enables spill code optimization and ultimate coalescing.

Approach. This paper assumes functions in Static Single Assignment (SSA) form after instruction selection as input. SSA functions are transformed to Linear SSA (LSSA) where each temporary is live in a single *basic block* and extended with optional copy instructions to support register allocation as in [3]. LSSA functions are augmented with *alternative temporaries*, a novel abstraction that supports the substitution of temporaries and enables spill code optimization and ultimate coalescing.

LSSA functions with alternative temporaries are transformed into combinatorial problems according to a formal model of register allocation and instruction scheduling which is parameterized with respect to a generic processor description. The model captures all major subproblems of global register allocation, including: spill code optimization and ultimate coalescing; multiple register banks; and register packing, where several small temporaries can be assigned to the same register. These subproblems are integrated with instruction scheduling and bundling for VLIW processors. The single model reflects the trade-off between interdependent register allocation and instruction scheduling decisions.

LCTES '14, June 12–13, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2597809.2597815

The combinatorial register allocation and instruction scheduling problems are solved using constraint programming [17], a technique that exploits the structure of the combinatorial model. Solutions to the combinatorial problems can be optimized accurately for different criteria such as speed, code size, or energy consumption. A presolving phase is introduced to increase robustness. Experiments on compiling medium-size MediaBench functions for Hexagon, a typical Digital Signal Processor (DSP), show that the introduced approach generates better code than existing combinatorial approaches and the LLVM code generator, and that the approach scales despite a significant growth of the solution space.

Contributions. The paper introduces a program representation and combinatorial model of register allocation and instruction scheduling that use the new concept of alternative temporaries to enable spill code optimization and ultimate coalescing. It shows presolving techniques that exploit the structure of the combinatorial model to increase robustness. Extensive experiments provide insight into the benefits and current limitations of the approach using Hexagon as a real-world DSP. The experiments demonstrate that the approach is robust, scales up to medium-size functions, adapts easily to different optimization criteria, and yields better code than previous combinatorial approaches and heuristic algorithms where LLVM is used as an example.

In the context of combinatorial optimization, the results are surprising. While the introduction of alternative temporaries leads to a combinatorial problem which is exponentially harder to solve, the approach is demonstrated to be scalable and robust while producing significantly better code.

Plan of the paper. Section 2 explains the necessary background. Alternative temporaries are introduced in Section 3, followed by the combinatorial model in Section 4. Unison as the code generator is discussed in Section 5. Section 6 contains the experimental evaluation. Related work is discussed in Section 7, and the paper concludes with Section 8.

2. Background

This section provides background information on combinatorial optimization, the program representation used in this paper, and the main aspects of register allocation.

2.1 Combinatorial Optimization

Register allocation and instruction scheduling are computationally hard combinatorial optimization problems. They can be solved by problem-specific heuristic algorithms (typically leading to suboptimal solutions) or by general combinatorial optimization techniques (potentially leading to optimal solutions). The latter amounts to *modeling* the problem and *solving* the resulting model with some combinatorial optimization technique.

A *combinatorial model* consists of *variables* (typically ranging over integers or Booleans), *constraints* expressing relations among the variables, and an *objective function* to define preferred solutions. A *solution* is a variable assignment satisfying all constraints of the model and an *optimal solution* minimizes (or maximizes) the value of the objective function. A combinatorial model serves as a template for an *instance*, which takes problem parameters into account resulting in a complete problem description.

Solving applies combinatorial optimization techniques to find solutions of an instance. Example techniques include constraint programming, integer programming, Boolean satisfiability solving , and local search. This paper uses *constraint programming* [17] as a modern combinatorial optimization technique. Constraint programming solvers interleave *constraint propagation* and *search* to find solutions. Constraint propagation discards values for variables (that is, partial assignments) that cannot be part of any solution. When no further propagation is possible, search tries several alternatives on which constraint propagation and search are repeated. Constraint propagation is essential to reduce the search space.

Constraint programming can capture and exploit explicit structure existing in many combinatorial models. Structure is captured by *global constraints* that express problem-specific relations among several variables. Global constraints offer two benefits: they ease modeling (as will become clear in Section 4) by capturing common patterns in problems, and enable strong propagation that leads to a drastically reduced search space.

2.2 Input Program

Instruction set. This paper assumes input functions for which Hexagon V4 instructions have been selected. Hexagon V4 is a DSP included in Qualcomm's Snapdragon system-on-chip for mobile devices [15]. Hexagon provides two typical embedded processor features: VLIW capabilities and different-width registers. VLIW processors exploit instruction-level parallelism by executing statically scheduled *bundles* of instructions in parallel. Hexagon bundles contain up to four 32-bit instructions. The register file includes 32 general-purpose registers (`R0`...`R31`) of 32 bits each which can be accessed as 16 registers (`R1:0`...`R31:30`) of 64 bits.

Control-flow graphs, operations, and temporaries. Functions are represented by their control-flow graph (CFG) in Static Single Assignment (SSA) form. A basic block (*block* for short) in the CFG consists of *operations* and *temporaries*. Operations use and define temporaries and are implemented by processor instructions. Temporaries are storage locations holding values corresponding to program variables after instruction selection. For example, an operation implemented by the Hexagon instruction `abs` that defines temporary t' as the absolute value of a used temporary t is denoted as $t' \leftarrow$ `abs` t. For clarity, examples in the paper only show instructions and temporaries that are relevant to the discussion.

Static Single Assignment form. SSA is a program form where temporaries are statically defined once. This form uses *ϕ-functions* at the join points of the CFG to merge definitions of the same temporary. This paper refers to SSA in its conventional form [19].

Liveness and interference. A *program point* is located between two consecutive statements. A temporary t is *live* at a program point if t holds a value that might be used in the future. The *live range* of a temporary t is the set of program points where t is live. The *basic definition of interference* states that two temporaries interfere if their live ranges overlap. The *ultimate notion of interference* by Chaitin *et al.* [4] additionally requires that the values of interfering temporaries differ. This refinement is essential for optimizing register allocation as is explained in Section 2.3.

Preassignments. Architectural constraints and application binary interfaces (ABIs) predetermine the registers to which certain temporaries must be assigned. A temporary t that is *preassigned* to a register `r` is denoted by $t \triangleright$`r`.

Example. Figure 1 shows the CFG of the factorial function in SSA form which is used as running example in the paper. The figure shows the purpose of ϕ-functions: for example, $t_{10} \leftarrow \phi(t_3, t_7)$ assigns t_{10} to either t_3 or t_7, depending on whether block b_3 is entered from b_1 or b_2. For readability, Hexagon register-to-register transfer (`tfr`), immediate transfer (`tfri`), load (`ldw`), store (`stw`), multiply (`mul`), indirect jump (`jump`), and conditional direct jump (`jump if`, `jump ifn`) instructions are shown in a simplified syntax. The top operation in b_1 and the bottom operation in b_3 define and use, respectively, the temporaries that are live on the function boundaries and are preassigned by the ABI: return address $t_1 \triangleright$`R31`, argument $t_2 \triangleright$`R0`, and return value $t_{10} \triangleright$`R0`.

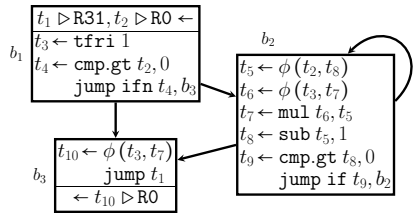


Figure 1. Factorial function in SSA with Hexagon instructions.

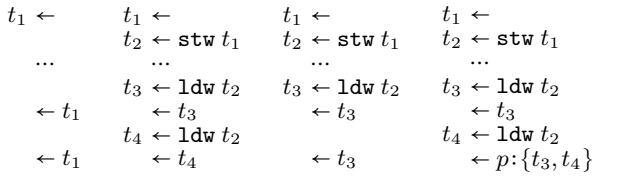


Figure 2. Spill code optimization.

2.3 Register Allocation

Register allocation assigns temporaries to either processor registers or memory. To reduce the number of memory accesses, registers are reused by non-interfering temporaries.

Spill code optimization. In general, optimal register utilization does not guarantee the availability of enough processor registers and some temporaries must be *spilled* (that is, stored in memory). Spilling a temporary requires the insertion of store and load instructions to move its value to and from memory. The selection and placement of these instructions has considerable impact on the efficiency of the generated code. In a *spill-everywhere* model, a load instruction is inserted immediately before each use of the spilled temporary. *Spill code optimization* reduces the cost of a spill by reusing the spilled temporary defined by a single load instruction.

Figure 2 shows an example where the temporary t_1 with two consecutive uses from Figure 2(a) is spilled to a new, memory-allocated temporary t_2 using a store instruction (`stw`) and reloaded using load instructions (`ldw`). In Figure 2(b) a load is inserted before each use, following the spill-everywhere model. In Figure 2(c), spill code optimization is performed to supply the two consecutive uses by the same load. Figure 2(d) is discussed in Section 3.

Coalescing. The input program may contain temporaries related by *copies* (operations that replicate the value of a temporary into another). Copy-related temporaries that do not interfere can be *coalesced* (assigned to the same register) in order to discard the corresponding copies and thereby improve efficiency and reduce code size. *Ultimate coalescing* considers all copy-related temporaries as candidates for coalescing: copy-related temporaries never interfere as they hold the same value.

The example in Figure 3 copies the value of t_1 into t_2 using a register-to-register transfer instruction (`tfr`). Using basic interference in Figure 3(a), t_1 and t_2 cannot be coalesced since their live ranges overlap. Using ultimate interference in Figure 3(b), t_1 and t_2 are coalesced into t_1 and the copy is discarded. Figure 3(c) is discussed in Section 3.

Packing. Each temporary t has a certain bit width (hereafter just called *width*) which is determined by t's source data type. Many processors allow temporaries of small widths to be assigned to different parts of a physical register of larger width. For example, the Hexagon processor combines pairs of 32-bits registers (`R3`, `R2`) into 64-bit registers (`R3:2`). Packing non-interfering temporaries into the same physical register is key to improving register utilization.

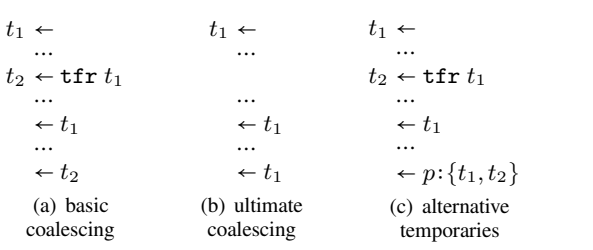


Figure 3. Coalescing.

Scope. *Local* register allocation deals with one block at a time, spilling all temporaries that are live at block boundaries. *Global* register allocation considers entire functions, yielding better code as temporaries can be kept in the same register across blocks.

2.4 Program Transformations

The combinatorial model described in Section 4 is based on a dedicated program representation for input functions. This representation uses the Linear Static Single Assignment form (LSSA) extended with copies as described by Castañeda *et al.* [3, Section 3] (note that they define *operation* and *instruction* reversely to this paper).

Linear Static Single Assignment form. Linear Static Single Assignment form (LSSA) is stricter than SSA in that each temporary is only defined and used within one block. The relation that two temporaries in different blocks correspond to the same temporary in SSA is captured by a *congruence* between temporaries. The local scope of LSSA temporaries leads to simple live ranges which is exploited in Section 4. This property also enables a decomposition scheme that can be exploited for robust code generation [3, Section 7]. LSSA is constructed from SSA by splitting each temporary t whose live ranges span multiple blocks $b_1, b_2, \ldots, b_n$ into a set of congruent local temporaries $t_{b_1} \equiv t_{b_2} \equiv \cdots \equiv t_{b_n}$. *Delimiter operations* are inserted at the boundaries of each block to define and use its live-in and live-out temporaries. These operations are not part of the generated code.

Copies. The LSSA program is extended with copies similarly to Appel and George's approach [1]. These operations are required to implement LSSA congruences and to handle spilling, multiple register banks, and preassignments.

A copy $t_d \leftarrow t_s$ replicates the value of the temporary t_s into the 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 can be implemented by alternative instructions $\{\bot, i_1, i_2, \ldots, i_n\}$ where the instruction i_j depends on the location types to which t_s and t_d are assigned. Copies implemented by the null instruction $\bot$ are discarded, otherwise they are *active* and must appear in the generated assembly code.

The copy insertion points and their alternative instructions depend on the processor. For example, Hexagon programs are extended with copies of the form $t_k \leftarrow \{\bot, \texttt{tfr}, \texttt{stw}\}\, t_s$ after the definition of t_s in a register and $t_d \leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, t_k$ before the use of t_d in a register, for all temporaries t_s, t_d not pre-assigned to a reserved register such as the return address register `R31`. Figure 5(a) shows the example given in Figure 2(b) where loads and stores are extended as copies that can be implemented by alternative instructions.

Example. Figure 4 shows the factorial function from Figure 1 in LSSA form extended with copies. Temporary t_1, whose live range spans all blocks in Figure 1, is split into the congruent local temporaries t_1, t_7, and t_{18} in Figure 4. The global relation between LSSA temporaries is solely captured by the congruences displayed on the arcs.

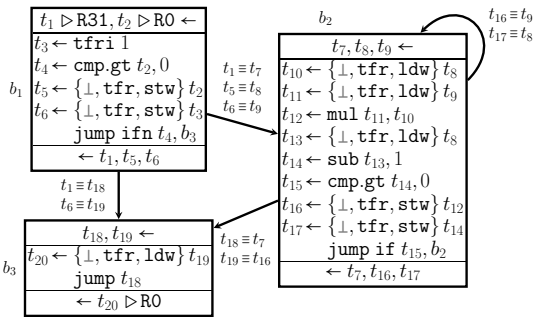


Figure 4. Function in LSSA extended with copies.

3. Alternative Temporaries

The program representation described in Section 2 yields a register allocation model that suffers from the limitations of spill-everywhere and basic coalescing [3, Section 6]. For example, optimizing the spill code in Figure 2(b) requires substituting the temporary t_3 for t_4 in the second use. Basic interference precludes ultimate coalescing as same-value temporaries cannot substitute each other. For example, coalescing temporaries t_1 and t_2 in Figure 3(a) requires substituting t_1 for t_2. In both cases, the program representation lacks the flexibility to substitute temporaries.

Alternative temporaries extend the capabilities of combinatorial register allocation by enabling spill code optimization and ultimate coalescing. This is achieved by augmenting the program representation with *operands* as use and definition ports in operations. Congruences, preassignments, and operation uses and definitions are lifted from temporaries to operands. Temporaries hold the values transferred among operations by *connecting* to the corresponding def- and use-operands. For example, the operation $t' \leftarrow$ abs t is transformed to $p' : t' \leftarrow$ abs $p : t$ where p and p' are operands connected to the temporaries t and t'. Operands can be connected to alternative temporaries that hold the same value to determine how the value is transferred among the corresponding operations. For example, if a second temporary t'' holds the same value as t, the operation above is transformed to $p' : t' \leftarrow$ abs $p : \{t, t''\}$ where either t or t'' can be connected to p. Examples in the paper omit operand identifiers when possible, for example as $t' \leftarrow$ abs $\{t, t''\}$.

Alternative temporaries enable the substitution of temporaries. For example, Figure 2(d) shows the alternative temporaries needed to optimize the spill code. If operand p is connected to temporary t_3, then t_4 is not used and the second load operation is discarded. As for the ultimate coalescing example, Figure 3(c) shows the alternative temporaries required to coalesce t_1 and t_2. If operand p is connected to temporary t_1, then t_2 is not used and the copy is discarded. In both cases, the proper temporary connections yield the intended results shown in Figures 2(c) and 3(b).

Construction. A program with alternative temporaries is constructed by replacing each occurrence of a temporary t with an operand p and a set of alternative temporaries that hold the same value as t and can thus be connected to p. This set can be effectively approximated by all temporaries that are copy-related to t. Figure 5(b) shows the alternative temporaries that correspond to the example given in Figure 5(a). This and all subsequent code examples are linearized for presentation purposes; no ordering among operations is imposed. For example, the second load operation in Figure 5(b) could actually be scheduled before the first one in the generated assembly code.

Discarding invalid connections. In the most general form, an operand that replaces a temporary t can be connected to any temporary that holds the same value as t. For example, operand p_1 in Figure 5(b) replaces an occurrence of t_2 in Figure 5(a) and can thus

$$
\begin{aligned}
t_1 &\leftarrow \\
t_2 &\leftarrow \{\bot, \texttt{tfr}, \texttt{stw}\}\, t_1 \\
t_3 &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, t_2 \\
&\leftarrow t_3 \\
t_4 &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, t_2 \\
&\leftarrow t_4
\end{aligned}
$$

(a) initial program

$$
\begin{aligned}
\{t_1, t_2, t_3, t_4\} &\leftarrow \\
\{t_1, t_2, t_3, t_4\} &\leftarrow \{\bot, \texttt{tfr}, \texttt{stw}\}\, \{t_1, t_2, t_3, t_4\} \\
p_2 : \{t_1, t_2, t_3, t_4\} &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, p_1 : \{t_1, t_2, t_3, t_4\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\} \\
p_3 : \{t_1, t_2, t_3, t_4\} &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, \{t_1, t_2, t_3, t_4\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\}
\end{aligned}
$$

(b) general form

$$
\begin{aligned}
t_1 &\leftarrow \\
t_2 &\leftarrow \{\bot, \texttt{tfr}, \texttt{stw}\}\, \{t_1, t_2, t_3, t_4\} \\
p_2 : t_3 &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, p_1 : \{t_1, t_2, t_3, t_4\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\} \\
p_3 : t_4 &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, \{t_1, t_2, t_3, t_4\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\}
\end{aligned}
$$

(c) single definitions

$$
\begin{aligned}
t_1 &\leftarrow \\
t_2 &\leftarrow \{\bot, \texttt{tfr}, \texttt{stw}\}\, t_1 \\
p_2 : t_3 &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, p_1 : \{t_1, t_2\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\} \\
p_3 : t_4 &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, \{t_1, t_2\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\}
\end{aligned}
$$

(d) acyclic connections

$$
\begin{aligned}
t_1 &\leftarrow \\
\{\bot, t_2\} &\leftarrow \{\bot, \texttt{tfr}, \texttt{stw}\}\, \{\bot, t_1\} \\
\{\bot, t_3\} &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, \{\bot, t_1, t_2\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\} \\
\{\bot, t_4\} &\leftarrow \{\bot, \texttt{tfr}, \texttt{ldw}\}\, \{\bot, t_1, t_2\} \\
&\leftarrow \{t_1, t_2, t_3, t_4\}
\end{aligned}
$$

(e) null connections

Figure 5. Step-by-step construction of alternative temporaries.

be connected to any of the same-value temporaries $\{t_1, t_2, t_3, t_4\}$. However, many of the potential combinations of connections in this form are invalid. For example, the def-operands p_2 and p_3 in Figure 5(b) cannot be connected to the same temporary t since that would define t twice. Likewise, the operands p_1 and p_2 cannot be connected to the same temporary since that would create a *connection cycle* where an operation uses its own definition. Such invalid connections are discarded to simplify the program representation as well as the combinatorial model in Section 4. The discarding process consists of two steps:

(1) *Single definitions* selects, for each temporary t, a single def-operand to which t can be connected. Figure 5(c) illustrates how enforcing single definitions prevents multiple definitions of the same temporary. For example, the two def-operands p_2 and p_3 in Figure 5(c) cannot be connected to the same temporary as is permitted in Figure 5(b). Enforcing single definitions discards some valid combinations such as connecting p_2 to t_4 and p_3 to t_3 in Figure 5(b). However, for each valid combination that is discarded a structurally equivalent permutation remains in the program representation. For example, connecting p_2 to t_3 and p_3 to t_4 is structurally equivalent to the combination mentioned above since t_3 and t_4 are interchangeable in all potential connections with use-operands.

(2) *Acyclic connections* discards combinations of temporary connections potentially leading to *connection cycles*. In a connection cycle, an operation o uses a temporary that depends on o's own

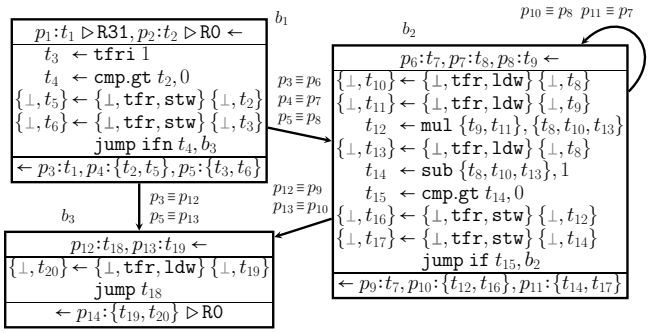


Figure 6. Function with alternative temporaries.

definition:

$$t_1 \leftarrow t_0;\ t_2 \leftarrow t_1;\ \ldots;\ t_n \leftarrow t_{n-1};\ t_0 \leftarrow t_n$$

Figure 5(d) illustrates how acyclic connections are enforced. For example, the operand p_1 in Figure 5(d) cannot be connected to temporary t_3 to create a cycle as is permitted in Figure 5(c). Enforcing acyclic connections preserves temporaries as alternatives in the operations where they originally appear. For example, temporary t_2 appears as a use of the second copy in Figure 5(a) and thus the potential connection of operand p_1 to t_2 is preserved in Figure 5(d). This property maintains the support gained by copy extension to handle problems such as spilling or preassignments.

Null connections. After invalid connections are discarded, null connections (denoted as $\perp$) are inserted as alternatives for copy operands to indicate that their corresponding copies can be discarded. A temporary potentially defined by a copy is *live* iff the copy is active as explained in Section 2.4. Figure 5(e) shows the final result after the insertion of null connections.

Example. Figure 6 shows the factorial function from Figure 4 augmented with alternative temporaries. For example, the second use-operand of the sub operation in block b_2 can be connected to three alternative temporaries: t_8, t_{10}, and t_{13}. These temporaries hold the same value since they are copy-related. The arc labels and delimiter operations illustrate how congruences and preassignments are lifted from temporaries to operands.

To summarize, alternative temporaries enable combinatorial spill code optimization and ultimate coalescing. The program representation includes operands as use and definition ports in operations. Alternative connections between operands and temporaries control the route followed by values among operations.

4. Combinatorial Model

This section incrementally describes the integrated combinatorial model for register allocation and instruction scheduling. The model introduces variables and constraints that capture alternative temporaries and enable spill code optimization and ultimate coalescing.

The combinatorial model is parameterized with respect to a program with alternative temporaries and a processor description. A program is described by the following parameters: a set of blocks B, operations O, operands P, and temporaries T; the function $\mathrm{operands}(o)$ that gives the operands of operation o; the function $\mathrm{temps}(p)$ that gives the temporaries that can be connected to operand p; and the predicate $\mathrm{use}(p)$ that indicates whether p is a use operand. Parameters that describe the processor and additional program parameters are introduced as needed.

4.1 Register Allocation

A solution to the register allocation problem corresponds to an assignment of the register allocation variables that satisfies Constraints 1-8 below. The register allocation variables are: a_o and i_o for each operation o to indicate whether o is *active* and which *instruction* implements it; l_t and r_t for each temporary t to indicate whether t is *live* and to which *register* it is assigned; and y_p for each operand p to indicate which *temporary* is connected to it.

The temporary connection variables and their related Constraints 1-4 are the essential improvements over the model presented by Castañeda *et al.* [3]. Each new y_p variable indicates which temporary is connected to the operand p among a set of alternative temporaries $\mathrm{temps}(p)$. Through these variables the combinatorial model can leverage the optimization opportunities enabled by the program representation with alternative temporaries. Although a new dimension of decision variables is introduced which yields an exponential growth of the solution space, the new model scales equally to that of Castañeda *et al.* as shown in Section 6.

Alternative temporaries and active operations. A temporary t is live iff it is used (that is, connected to some use-operand p):

$$l_t \iff \exists p \in P : (\mathrm{use}(p) \wedge y_p = t) \quad \forall t \in T \tag{1}$$

The *single definitions* step in the construction of alternative temporaries forces each temporary t to be connectable to exactly one def-operand. The operation that contains this operand is given by $\mathrm{definer}(t)$ and is active iff t is live:

$$a_{\mathrm{definer}(t)} \iff l_t \quad \forall t \in T \tag{2}$$

Active operations are connected to temporaries and are implemented by non-null instructions:

$$a_o \iff y_p \neq \perp \quad \forall o \in O,\ \forall p \in \mathrm{operands}(o) \tag{3}$$

$$a_o \iff i_o \neq \perp \quad \forall o \in O \tag{4}$$

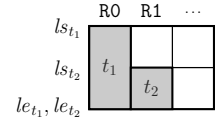
Unified register array and instruction selection. The combinatorial model is based on a unified register array where its elements can be either processor registers from different banks or a practically infinite number of *memory registers* (representing memory locations on the runtime stack) [3, Section 4.1]. In this abstraction memory registers form a *register class* in the same way as the rest of register subsets that are interchangeable for some instruction.

Instructions access their operands in certain register classes as determined by the processor. The processor parameter $\mathrm{class}(\mathtt{i}, p)$ gives the register class of operand p if its operation is implemented by instruction i. The instruction that implements an operation determines the register class to which its operands are allocated:

$$r_{y_p} \in \mathrm{class}(i_o, p) \quad \forall o \in O,\ \forall p \in \mathrm{operands}(o) \tag{5}$$

Register assignment and packing. Register assignment maps non-interfering temporaries to registers. The combinatorial model captures register assignment according to the basic definition of interference. It relies on alternative temporaries to avoid solutions where same-value temporaries are simultaneously live, effectively delivering ultimate coalescing. This yields a simple geometric representation of register assignment that can be exploited by existing global constraints in constraint programming.

Local register assignments can be projected onto a rectangular area as described by Castañeda *et al.* [3, Section 4.1] and illustrated by the figure to the left. The horizontal dimension represents the registers in the unified register array, and the vertical dimension represents time in clock cycles. Each live temporary t yields a rectangle with $\mathrm{width}(t) = 1$ (the width is later redefined for register packing). The top and bottom coordinates of t reflect the live start (ls_t) and end (le_t) cycles, which correspond to the issue cycles of t's definer and last user. The horizontal coordinate represents the register



to which the temporary is assigned. For register allocation in isolation, live ranges are given as program parameters. For the model extended with instruction scheduling these are variables that depend on the computed schedule as described in Section 4.2.

In this representation, two live temporaries (for example, t_1 and t_2 above) interfere iff their rectangles overlap vertically. The non-overlapping rectangles constraint $\mathrm{disjoint2}$ [17] forces such interfering temporaries to be assigned to different registers:

$$\mathrm{disjoint2}(\{\langle r_t, r_t{+}\mathrm{width}(t) \times l_t, ls_t, le_t\rangle{:}t{\in}\mathrm{T}(b)\}) \quad \forall b{\in}B \quad (6)$$

where the program parameter $\mathrm{T}(b)$ gives the temporaries in block b.

As Section 2 explains, certain operands are preassigned to registers. The temporaries connected to such operands are preassigned to the corresponding registers:

$$r_{y_p} = \mathtt{r} \quad \forall p{\in}P : p \rhd \mathtt{r} \quad (7)$$

Register packing is readily captured by this representation as explained by Castañeda *et al.* 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, `R5` in Hexagon). Each column in the unified register array corresponds to an atom, where atoms representing different parts of a larger register are adjacent. $\mathrm{width}(t)$ is redefined as a processor parameter giving the number of atoms that temporary t occupies. The variable r_t represents the first of the atoms to which t is assigned. Enforcing non-interference among live temporaries assigned to the same register (Constraint 6) thus becomes isomorphic to rectangle packing.

Global Register Allocation. In LSSA with alternative temporaries, blocks are solely related by operand congruences, which leads to a direct extension of the local problem [3, 20]. The program parameter $p \equiv q$ indicates that operands p and q are congruent. Congruent operands are assigned to the same register:

$$r_{y_p} = r_{y_q} \quad \forall p, q{\in}P : p \equiv q \quad (8)$$

4.2 Instruction Scheduling and Bundling

An *issue cycle* variable c_o is defined for each operation o. A solution to the instruction scheduling problem corresponds to an assignment of the issue cycle variables that satisfies Constraints 9-12 below. Instruction bundling for VLIW processors such as Hexagon is captured by interpreting sets of operations issued in the same cycle as bundles. The live start (ls_t) and end (le_t) of a temporary t are variable in the integrated register allocation and instruction scheduling problem, as they depend on the issue cycle of the definer and users of t. More specifically, the live range of a temporary t starts at the issue cycle of its definer:

$$l_t \implies ls_t = c_{\mathrm{definer}(t)} \quad \forall t{\in}T \quad (9)$$

and ends with the last issue cycle of its users:

$$l_t \implies le_t = \max_{o{\in}\mathrm{users}(t)} c_o \quad \forall t{\in}T \quad (10)$$

where $\mathrm{users}(t)$ gives the operations that contain a use-operand connected to t.

If a temporary t is not used and hence not live (Constraint 1) its live start and end variables are unconstrained. This does not compromise the correctness of the model since these variables only matter if t is live (Constraint 6).

Precedences. The processor parameter $\mathrm{lat}(\mathtt{i})$ gives the latency with which instruction `i` defines its resulting temporaries. An operation that uses a temporary t can only be issued after the issue cycle plus the latency of the definer of t:

$$a_o \implies c_o \geq c_{\mathrm{definer}(y_p)} + \mathrm{lat}(i_{\mathrm{definer}(y_p)}) \quad \forall o{\in}O,\ \forall p{\in}\mathrm{operands}(o) : \mathrm{use}(p) \quad (11)$$

Processor resources. Operations share limited processor resources such as functional units and data buses. They are described by the following processor parameters: a set of resources R; the functions $\mathrm{con}(\mathtt{i},\mathtt{r})$ and $\mathrm{dur}(\mathtt{i},\mathtt{r})$ that give the units of a resource `r` consumed by instruction `i` and the cycles during which `r` is consumed; and the function $\mathrm{cap}(\mathtt{r})$ that gives the capacity of resource `r` in number of units. The capacity of each processor resource cannot be exceeded at any issue cycle. This structure is naturally captured as a task-resource model with a $\mathrm{cumulative}$ constraint [17] for each block and processor resource. Each $\mathrm{cumulative}$ constraint includes a task for each operation o in the block where the consumption and duration are zero if o is implemented by the null instruction:

$$\mathrm{cumulative}(\{\langle c_o, \mathrm{con}(i_o,\mathtt{r}), \mathrm{dur}(i_o,\mathtt{r})\rangle : o{\in}\mathrm{O}(b)\}, \mathrm{cap}(\mathtt{r})) \quad \forall b{\in}B, \forall \mathtt{r}{\in}R \quad (12)$$

where the program parameter $\mathrm{O}(b)$ gives the operations in block b.

4.3 Optimization criteria

Code generation typically aims at solutions that are as good as possible for some optimization criterion such as speed, code size, or energy consumption. Traditional heuristic algorithms embed such optimization criteria implicitly into decisions. For example, traditional list scheduling [16] issues operations as early as possible under the implicit assumption that this yields compact schedules. Unlike heuristic algorithms, the model captures different optimization criteria accurately and unambiguously in a generic minimization objective function:

$$\sum_{b{\in}B} \mathrm{weight}(b) \times \mathrm{cost}(b)$$

where $\mathrm{weight}(b)$ and $\mathrm{cost}(b)$ give the weight and estimated cost of block b. Note that also non-linear objective functions are possible but for the purpose of this paper linear functions are sufficient. To optimize for speed, $\mathrm{weight}(b)$ is set to $\mathrm{freq}(b)$ (a program parameter giving the estimated execution frequency of block b), and $\mathrm{cost}(b)$ is defined as $\max_{o{\in}\mathrm{O}(b):a_o} c_o$. To optimize for code size, $\mathrm{weight}(b)$ is disregarded and $\mathrm{cost}(b)$ is defined as $\sum_{o{\in}\mathrm{O}(b)} \mathrm{con}(i_o, \mathtt{bits})$, where the processor resource `bits` represents the bits with which instructions are encoded. Optimization criteria such as energy consumption can be modeled analogously.

4.4 Limitations

While the introduced combinatorial model captures a wide array of register allocation and instruction scheduling subproblems (as demonstrated in Section 7), it still exhibits some limitations to be addressed in the future.

Unpredictable processor features like cache memories lead to instruction latencies which are unknown at compilation time. As is common in combinatorial approaches, the introduced model assumes the best-case for such latencies and relies on pipeline stalling to handle worse cases. This assumption may underestimate the contribution of unknown latencies to the objective function.

The introduced model does not permit to move operations across blocks, which limits the amount of exploitable instruction-level parallelism. Existing combinatorial models of global instruction scheduling could be integrated with our approach [14, 22].

In some cases it is beneficial to recompute (that is, *rematerialize*) a reused value rather than occupying a register until its later use, or spilling. The model does not currently support rematerialization, but alternative temporaries may allow its incorporation following the approach of Goodwin and Wilken [10].

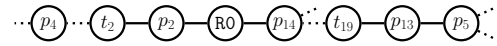


Figure 7. Subgraph of the connection graph G for Figure 6.

5. Code Generator

This section describes *Unison*, a constraint-based code generator for the model introduced in Section 4 that features robust and scalable optimization by using presolving and decomposition techniques that exploit properties of the program representation. Unison is implemented on top of the constraint programming system Gecode [9].

Presolving. Presolving techniques reformulate combinatorial problems to boost the robustness of the solving process. Unison uses an array of dedicated presolving techniques, including: generating *implied constraints* (logically redundant constraints that improve propagation), computing lower bounds by solving *problem relaxations* where some constraints are excluded, and detecting and removing redundant solutions to avoid unnecessary search.

A common approach to generate implied constraints is by negating *nogoods* – combinations of variable assignments that cannot hold together. This paper introduces *connection nogoods* as a particularly effective presolving technique. A connection nogood is a conjunction of connections of the form $y_p = t$. The generation process consists of two steps:

(1) A *connection graph* G (Figure 7 shows an example) is constructed from a LSSA function with alternative temporaries. G contains a node for each operand, temporary, and pre-assigned register, and two classes of edges: *must-connect* edges (solid) relating congruent operands, operands and registers in preassignments, and operands and temporaries whose connection is forced by single-alternatives; and *may-connect* edges (dotted) relating operands with the temporaries that may be connected to them.

(2) Nogoods are derived by analyzing G as follows. Two distinct nodes *interfere* if both of them are either registers, or use- or def-operands of the same operation. For example, p_4 and p_5 in Figure 7 interfere since both are use-operands of the bottom delimiter operation of b_1. Interfering nodes cannot be transitively connected. Suppose that there is a path in G between two interfering nodes that crosses k may-connect edges $(p_1, t_1), \ldots, (p_k, t_k)$. Then the problem has no solution if the operands and temporaries in all k may-connect edges in the path are connected. This derives $y_{p_1} = t_1 \wedge \ldots \wedge y_{p_k} = t_k$ as a connection nogood.

For example, in Figure 7 there is a path between p_4 and p_5 that crosses the may-connect edges (p_4, t_2) and (p_{14}, t_{19}), yielding the nogood $y_{p_4} = t_2 \wedge y_{p_{14}} = t_{19}$.

Problem decomposition. Unison exploits properties of LSSA to decompose the problem as introduced in Castañeda *et al.* [3], which increases its scalability and gives it *anytime behavior* – solutions are found in increasing quality as code generation runs.

LSSA temporaries are live in single blocks and only indirectly related to temporaries from other blocks by congruences on *global* operands. Global operands belong to the delimiter operations inserted during LSSA construction. For example, the global operands in Figure 6 are $\{p_1, p_2, \ldots, p_{14}\}$. The register variables of these operands are related by congruence constraints in the combinatorial model (Constraint 8). Once these variables are assigned, the rest of the register allocation and instruction scheduling problem can be solved independently for each block.

Unison exploits this observation to proceed iteratively as shown by Figure 8: (1) A SSA function is transformed to LSSA, extended with copies, and augmented with alternative temporaries; (2) the function is translated into a problem according to the model introduced in Section 4 and presolved with the techniques introduced above; (3) a *global problem* is solved by assigning the register variable r_{y_p} of each global operand p such that all constraints are satisfied. Search in the global solver is arranged in two phases: *global allocation* and *global assignment*, under a global time limit. Global allocation selects a register class (possibly memory) for each set of congruent global operands G in decreasing size of G. A cost-benefit analysis determines the register class to which the selected set G is allocated first. The benefit component estimates the saved spilling overhead, while the cost component is based on an estimate of the increased register pressure. This analysis is parameterized with an *aggressiveness* factor to guide the allocation towards either the benefit or the cost component. Global assignment selects a particular register from the register class allocated to each set G; (4) a *local problem* is solved for each block b by assigning its remaining variables such that all but the congruence constraints are satisfied and $\mathrm{cost}(b)$ is minimized. A *search portfolio* is used to increase the robustness of the local solver, where up to five complementary search strategies are applied sequentially until an optimal local solution is found or a local time limit is reached. All search strategies are arranged in multiple phases, typically starting with the active (a_o), instruction (i_o) and temporary (y_p) variables; following with the issue cycle variables (c_o); and finishing with the register variables (r_t). (5) the solutions to the global and local problems are combined into a full solution s and a new iteration is run from (3), increasing the aggressiveness of the global solver and constraining future solutions to be better than s according to the objective function in Section 4.3; and (6) when optimality is proven or a time limit is reached, assembly code is generated according to the last full solution (which, by construction, is the best one found).

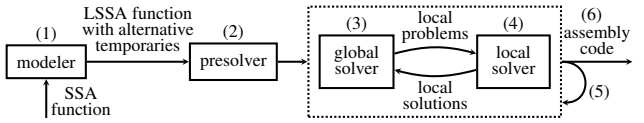


Figure 8. Architecture of the *Unison* code generator.

6. Experimental Evaluation

This section presents experimental results on different characteristics of Unison: code quality improvements; the impact of using alternative temporaries and presolving techniques; scalability and runtime behavior; and using different optimization criteria.

Setup. As input for the experiments we use MediaBench, a benchmark suite widely employed in embedded compiler research [13]. Ten medium-size functions (from 25 to 1000 instructions) are sampled from each signal-processing application included in the benchmark suite (`jpeg`, `mpeg`, `gsm`, `g721`, `epic`, `adpcm`), with the exception of `adpcm` where only five functions are available. The purpose of sampling is to shorten the runtime of the experimental evaluation while conserving a set of functions that is representative of the benchmark. Cluster sampling is applied on each application by randomly selecting a function from each cluster, computed by a 10-means analysis. Functions are clustered by size (in number of input LLVM operations) and register pressure (approximated as the fraction of temporaries spilled by LLVM's register allocator).

Each function is compiled and optimized using the LLVM 3.3 compiler infrastructure with the `-O3` flag, and Hexagon V4 instructions are selected using LLVM's instruction selector. Due to limitations in the current interface between our prototype and LLVM, certain low-level CFG and alias analysis optimizations in LLVM are disabled to ensure an accurate comparison. However, these optimizations are all orthogonal to the combinatorial model itself; in fact, disabling them is disadvantageous to the constraint-based approach which has more potential to lever-

age instruction-level parallelism by considering the full solution space. The following flags, prefixed with `-disable-`, are used: `post-ra`, `tail-duplicate`, `branch-fold`, `block-placement`, `phi-elim-edge-splitting`, and `hexagon-cfgopt`.

The evaluation uses the number of execution cycles (*cycles* for short) of a generated assembly function as a measure of its quality. This number is estimated statically according to the speed criterion defined in Section 4.3; lack of post-code-generation support and limited access to Hexagon development tools prevent us from measuring the actual number of execution cycles. The execution frequency freq is estimated by LLVM's code generator.

The global and local solvers in Unison are implemented on top of the constraint programming system Gecode 4.2.1. On each function Unison runs for ten iterations (the decision for this number will become clear when discussing the runtime behavior). Every iteration has a global time limit of $7 * i$ ms, where i is the number of operations in the function, and a local time limit of 25 s. The experiments are run with a single thread on a Linux machine equipped with an Intel Xeon E5607 2.27 GHz processor and 24 GB of RAM. All results are averaged over 10 repetitions. The maximum coefficient of variation for the average solving time and code quality (speed or code size) for all functions is 1.6% and 0.3%, respectively.

Code quality compared to traditional approaches. We compare the quality of the code generated by Unison with that of the code generated by LLVM as a representative of traditional, staged approaches. LLVM solves register allocation and local instruction scheduling by priority-based coloring [6] and list scheduling [16]. Global instruction scheduling is not yet available in LLVM.

The results indicate that our approach delivers code of significantly better quality than LLVM for the MediaBench sample. Figure 11(a) shows the cycle improvement using our approach over LLVM for each of the 55 functions. Unison improves code quality for 39 functions (up to 40.9%), and produces inferior code for 7 functions (down to -7.0%), yielding a geometric mean (GM) improvement of 7.1%. Optimal solutions are found for 16 functions.

The best cases typically correspond to functions that contain large blocks and multiple function calls. Unison can efficiently handle the former by advanced VLIW bundling and the latter by integrated handling of ABI preassignments, packing, and spill code optimization. Three of the worst cases (for `run_length_decode_zeros`, `decode_mcu`, and `decode_mcu_AC_first`) are affected by the lack of rematerialization in the combinatorial model, as discussed in Section 4.4. This limitation explains why Unison's solution for `run_length_decode_zeros` is optimal but worse than LLVM's. The remaining inferior cases are due to incomplete communication between the global and the local solver of the decomposition introduced in Section 5.

Impact of alternative temporaries on code quality. Alternative temporaries allow the generation of better code by enabling spill code optimization and ultimate coalescing. To measure this benefit we compare the cycles of optimal solutions found for the model with alternative temporaries against those found for the model without from Castañeda *et al.* [3].

The comparison demonstrates that alternative temporaries have indeed a positive impact on code quality. Figure 9 shows the cycle improvement given by the model with alternative temporaries, for the functions solved to optimality. In the cases indicated with vertical dots only a lower bound of the improvement could be computed. The use of alternative temporaries improves code quality for 16 out of the 26 functions solved to optimality (up to 10.8%), yielding a GM improvement of 2.2%. No solution is worse, which confirms the hypothesis that using alternative temporaries can only improve code quality.

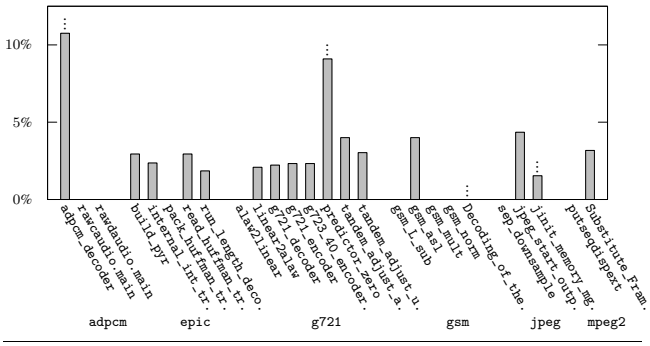


Figure 9. Cycle improvement over the model from Castañeda *et al.* for optimal solutions. Vertical dots indicate lower bounds.

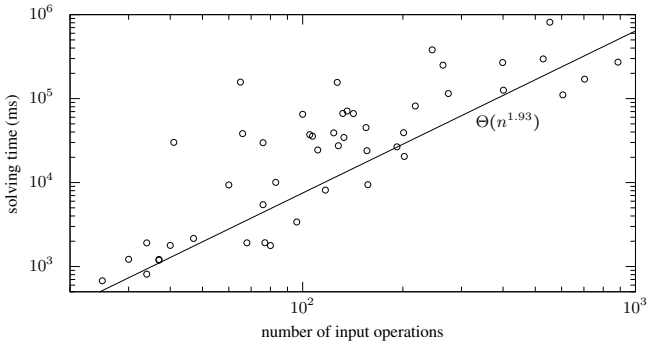


Figure 10. Time to reach LLVM code quality vs. input size.

Impact of presolving on robustness. Section 5 introduces presolving techniques to make code generation more robust. To measure the impact of these techniques we compare the results obtained in the first experiment (*code quality compared to traditional approaches*) with a similar experiment where presolving is disabled.

The results confirm that the presolving techniques are indeed essential for Unison's robustness: without them, 6 out of the 55 functions cannot be solved at all, and the code quality of the solved functions decreases drastically. The number of functions for which Unison improves code quality drops from 39 to 28 (up to 40.9% for `predictor_zero`) and the number of inferior cases grows from 7 to 15 (down to -40.4% for `g723_40_encoder`). The GM improvement over LLVM decreases considerably, from 7.1% down to 0.2%. Also, only 8 optimal solutions can be found.

Scalability. To quantify the scalability of Unison we measure the solving time to generate code that is on par with LLVM in terms of cycles. For each function, Unison iterates until the solution quality is at least as good as that of LLVM. The 7 functions which cannot reach the baseline quality are excluded from the experiment.

The measurements demonstrate that Unison is scalable. Figure 10 shows the solving time to generate as good code as LLVM. The figure reveals an average computational complexity that is approximately quadratic in the number of input operations – a least square analysis gives $\Theta(n^{1.93})$. This scalable behavior is due to the combination of decomposition, presolving, and time limits.

A similar figure is presented by Castañeda *et al.* which suggests that their code generator can handle functions approximately four times larger, but they measure the number of operations after the input is extended with copies, a process that inflates the size by an average factor of 3.1. Furthermore, they only report results for MIPS32 – a simple, single-issue processor. In other words, even with spill code optimization and ultimate coalescing our model scales as well as the model of Castañeda *et al.* Compared to LLVM, however, Unison is still orders of magnitude slower.

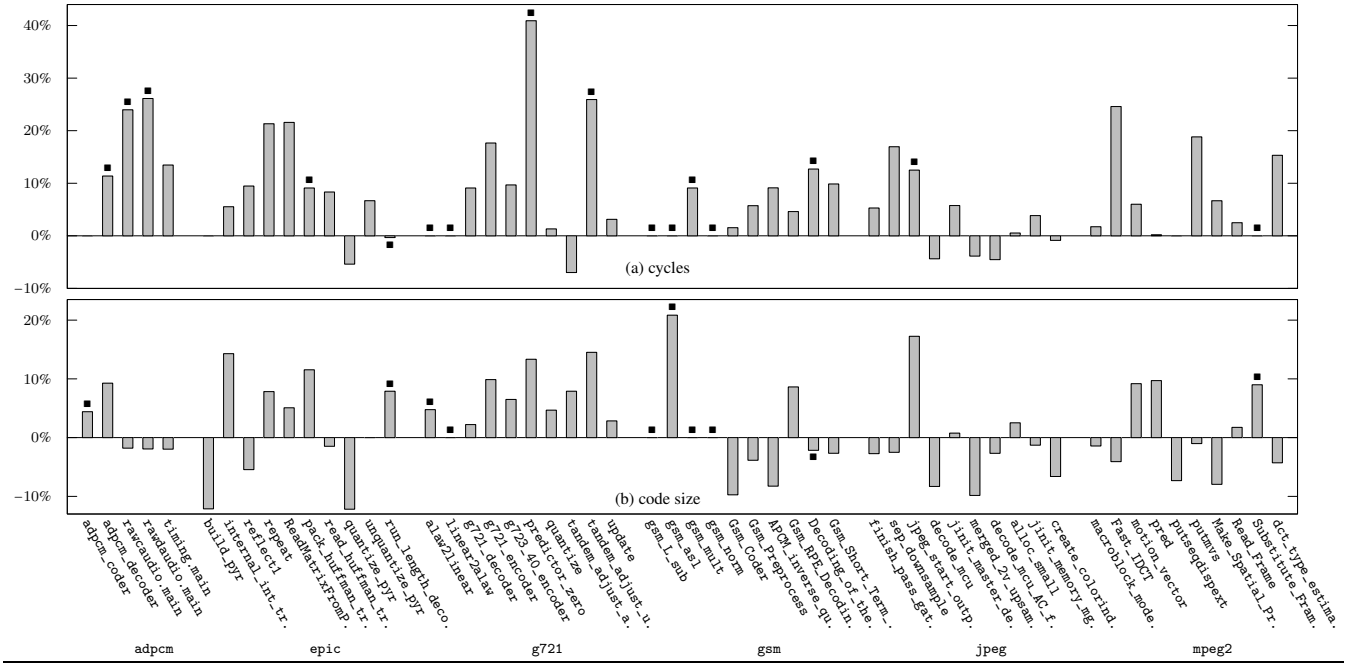


Figure 11. Cycle and code size improvement over LLVM. Negative bars means that results are worse, and ■ indicates optimal solutions.

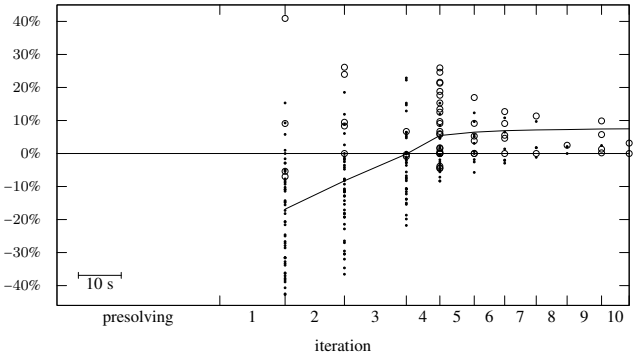


Figure 12. Cycle improvement over LLVM per iteration. In every iteration a dot corresponds to the best solution found per function. When a function cannot be improved further by Unison, the last solution is indicated by a circle in the iteration it was found. The curve shows the cycle improvement across all functions. The horizontal range is proportional to the average time of each iteration.

Runtime behavior. To study how Unison behaves as it runs, we measure the time spent and its quality improvement over LLVM for each iteration. The results indicate that Unison has a reasonable anytime behavior, in which each iteration progressively delivers better code. Figure 12 demonstrates this runtime behavior. Three stages can be identified in an average run of Unison: first, presolving together with the first iteration deliver an initial solution (taking 39.9% of the total runtime). This delay is mostly due to presolving; then, iterations two to four drastically improve the code quality (7.4% in average, taking 27.0% of the total runtime); finally, the code quality appears to converge, and the remaining iterations only improve the code quality marginally (0.4% in average, taking the remaining 28.3% of the runtime). This convergent behavior motivates running ten iterations for the experiments.

Impact of different optimization criteria on robustness. The combinatorial model can be easily adjusted to optimize for different criteria, as described in Section 4.3. To study the robustness of Unison for different optimization criteria, we switch to code size optimization and compare the size of the code generated by Unison with that of the code generated by LLVM. The LLVM intermediate optimizer and code generator are run with the flags `-Oz` and `-O2`, respectively. The latter is chosen as the highest optimization level that does not trade larger code size for higher speed since LLVM's code generator does not support explicit code size optimization.

The results demonstrate that Unison robustly adapts to different optimization criteria. Figure 11(b) shows the size improvement using our approach over LLVM for each function. Unison is already competitive with LLVM even though no effort has been invested in tuning it for code size optimization. Among the 55 functions our approach improves code size for 25 functions (up to 20.8%), and produces larger code size for 25 functions (down to −12.2%), yielding a slight GM improvement of 1.2%. Optimal solutions are found for 10 functions. As above, `Decoding_of_the_coded_Log_Area_Ratios` is optimal because Unison's model does not include rematerialization.

7. Related Work

There is a significant body of research on solving the three main code generation problems (instruction selection, register allocation, and instruction scheduling) with combinatorial optimization techniques, both with integrated models and in isolation. Table 1 summarizes the characteristics of the most prominent approaches that handle register allocation. The approaches are classified into three groups: those which also integrate instruction scheduling and selection (top), instruction scheduling only (middle); or solve register allocation in isolation (bottom).

Among the integrated approaches (top and middle), only those of Wilson *et al.* [20], Castañeda *et al.* [3], and this paper support global register allocation. The three approaches handle it by constraints that assign related, inter-block temporaries to the same register. Furthermore, this paper is the first integrated approach that features ultimate coalescing and, together with Castañeda *et al.*, the only integrated approach that features register packing.

approach	TC	SO	CO	GL	RP	MB	RM	SC	SL
Wilson [20]	IP	-	basic	✓	-	-	-	✓	✓
Gebotys [8]	IP	✓	-	-	-	✓	-	✓	✓
Bashford [2]	CP	✓	basic	-	-	✓	-	✓	✓
Eriksson [7]	IP	-	-	-	-	✓	-	✓	✓
Chang [5]	IP	✓	-	-	-	-	-	✓	-
Kästner [11]	IP	-	-	-	-	-	-	✓	-
Nagarakatte [14]	IP	✓	-	-	-	-	-	✓	-
Castañeda [3]	CP	-	basic	✓	✓	✓	-	✓	-
(this paper)	**CP**	✓	**ultimate**	✓	✓	✓	-	✓	-
Goodwin [10]	IP	✓	basic	✓	✓	-	✓	-	-
Appel [1]	IP	✓	-	✓	-	-	-	-	-
Scholz [18]	PBQP	-	ultimate	✓	✓	-	-	-	-
Krause [12]	DP	✓	basic	✓	✓	-	✓	-	-

Table 1. Related combinatorial approaches: *TeChnique (Integer, Constraint, Partitioned Boolean Quadratic, and Dynamic Programming)*, *Spill code Optimization*, *COalescing*, *GLobal register allocation*, *Register Packing*, *Multiple register Banks*, *ReMaterialization*, *instruction SCheduling*, and *instruction SeLection*.

Isolated register allocation approaches (bottom) exploit knowledge about the selected instructions and the computed schedule to formulate simpler models and capture more subproblems than the integrated ones. For example, Scholz and Eckstein's model derives constraints from temporary interferences [18], which is only possible if an instruction schedule is assumed. It is worth noticing that this paper only lacks rematerialization to match the features of the isolated approaches. All other integrated approaches provide at least two features less than their isolated counterparts.

The combinatorial model introduced in this paper builds on the work by Castañeda *et al.* by incorporating spill code optimization and ultimate coalescing, while retaining the original robustness for medium-size functions. Furthermore, while Castañeda *et al.* demonstrate the capabilities of their approach on a simple MIPS32 processor this paper reports results for Hexagon, a more challenging processor with VLIW capabilities and different-width registers.

The concept of alternative temporaries is related to the idea of *alternative implementations* by Wilson *et al.* [20]. Alternative implementations consist of groups of operations among which the integer programming solver must choose one for execution. This abstraction is exploited to generate spill code and register-to-register copies, and to select array addressing instructions. Unfortunately, the publications by Wilson *et al.* [20, 21] do not provide enough detail to determine to which extent *alternative temporaries* and *alternative implementations* are related.

8. Conclusion and Future Work

This paper has introduced a program representation and combinatorial model of register allocation and instruction scheduling that use alternative temporaries to enable spill code optimization and ultimate coalescing. The model is shown to be scalable and robust while matching, for the first time, the features of traditional heuristic approaches. Thorough experiments on a real-world DSP demonstrate that the approach can be easily adapted to different optimization criteria and generates better code than heuristics and previous combinatorial approaches.

Future work. There is considerable future work towards combinatorial code generation. A first step is to address the limitations identified in Section 4.4. Furthermore, the runtime behavior can be improved by integrating different solving techniques, including randomization and search restarts for robustness, large neighborhood search for scalability, and computation of stronger bounds with integer programming for shorter solving time.

Acknowledgments

This research 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 Frej Drejhammar, Peter A. Jonsson, Ingo Sander, and the anonymous reviewers.

References

[1] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. *SIGPLAN Not.*, 36:243–253, May 2001.

[2] S. Bashford and R. Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems*, pages 119–165, Mar. 1999.

[3] R. Castañeda Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. Constraint-based register allocation and instruction scheduling. In *CP*, volume 7514 of *LNCS*, pages 750–766. Springer, 2012.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[5] C.-M. Chang, C.-M. Chen, and C.-T. King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers Math. Applic.*, 34:1–14, Nov. 1997.

[6] F. Chow and J. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Not.*, 19(6):222–232, June 1984.

[7] M. V. Eriksson, O. Skoog, and C. W. Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *SCOPES*, 2008.

[8] C. H. Gebotys. An efficient model for DSP code generation: Performance, code size, estimated energy. In *ISSS*, pages 41–47. IEEE, 1997.

[9] Gecode Team. Gecode: generic constraint development environment. `www.gecode.org`, 2006.

[10] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Software – Practice and Experience*, 26:929–965, Aug. 1996.

[11] D. Kästner. PROPAN: A retargetable system for postpass optimisations and analyses. In *LCTES*, volume 1985 of *LNCS*, 2001.

[12] P. K. Krause. Optimal register allocation in polynomial time. In *CC*, volume 7791 of *LNCS*, pages 1–20. Springer, 2013.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO-30*, pages 330–335. IEEE, 1997.

[14] S. G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *CC*, volume 4420 of *LNCS*, pages 126–140. Springer, 2007.

[15] *Hexagon V4 Programmer's Reference Manual*. Qualcomm Technologies, Inc., Aug. 2013.

[16] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *J. Supercomput.*, 7:9–50, May 1993.

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

[18] B. Scholz and E. Eckstein. Register allocation for irregular architectures. *SIGPLAN Not.*, 37:139–148, June 2002.

[19] V. Sreedhar, R. Ju, D. Gillies, and V. Santhanam. Translating out of static single assignment form. In *SAS*, volume 1694 of *LNCS*, pages 849–849. Springer, 1999.

[20] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An integrated approach to retargetable code generation. In *ISSS*, pages 70–75. IEEE, 1994.

[21] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-based approach to code generation. In *Code Generation for Embedded Processors*, pages 103–118. Springer, 2002.

[22] S. Winkel. Optimal versus heuristic global code scheduling. In *MICRO-40*, pages 43–55. IEEE, 2007.