Combinatorial Register Allocation and Instruction Scheduling

ROBERTO CASTAÑEDA LOZANO, RISE SICS, Sweden and KTH Royal Institute of Technology, Sweden

MATS CARLSSON, RISE SICS, Sweden

GABRIEL HJORT BLINDELL, KTH Royal Institute of Technology, Sweden

CHRISTIAN SCHULTE, KTH Royal Institute of Technology, Sweden and RISE SICS, Sweden

This paper introduces a combinatorial optimization approach to register allocation and instruction scheduling, two central compiler problems. Combinatorial optimization has the potential to solve these problems optimally and to exploit processor-specific features readily. Our approach is the first to leverage this potential *in practice*: it captures the *complete* set of program transformations used in state-of-the-art compilers, *scales* to medium-sized functions of up to 1000 instructions, and generates *executable* code. This level of practicality is reached by using constraint programming, a particularly suitable combinatorial optimization technique. Unison, the implementation of our approach, is open source, used in industry, and integrated with the LLVM toolchain.

An extensive evaluation confirms that Unison generates better code than LLVM while scaling to medium-sized functions. The evaluation uses systematically selected benchmarks from MediaBench and SPEC CPU2006 and different processor architectures (Hexagon, ARM, MIPS). Mean estimated speedup ranges from 1.1% to 10% and mean code size reduction ranges from 1.3% to 3.8% for the different architectures. A significant part of this improvement is due to the integrated nature of the approach. Executing the generated code on Hexagon confirms that the estimated speedup results in actual speedup. Given a fixed time limit, Unison solves optimally functions of up to 946 instructions, nearly an order of magnitude larger than previous approaches.

The results show that our combinatorial approach can be applied in practice to trade compilation time for code quality beyond the usual compiler optimization levels, identify improvement opportunities in heuristic algorithms, and fully exploit processor-specific features.

CCS Concepts: • **Computing methodologies** → **Discrete space search**; *Planning and scheduling*; • **Software and its engineering** → **Compilers**; *Constraint and logic languages*.

Additional Key Words and Phrases: combinatorial optimization; register allocation; instruction scheduling

1 INTRODUCTION

Register allocation and instruction scheduling are central compiler problems. Register allocation maps temporaries (program or compiler-generated variables) to registers or memory. Instruction scheduling reorders instructions to improve total latency or throughput. Both problems are key to generating high-quality assembly code, are computationally complex (NP-hard), and are mutually

interdependent: no matter in which order these problems are approached, aggressively solving one of them might worsen the outcome of the other [45, 47, 49, 74].

Today's compilers typically solve each problem in isolation with heuristic algorithms, taking a sequence of greedy decisions based on local optimization criteria. This arrangement reduces compilation time but precludes optimal solutions and complicates the exploitation of processor-specific features.

Combinatorial optimization has been proposed to address these limitations: it can deliver optimal solutions according to a model, it can accurately capture problem interdependencies, and it can accommodate specific architectural features at the expense of increased compilation time [24]. While this approach shows promise for register allocation and instruction scheduling, its practical significance has not yet been established. Typically, combinatorial approaches model only a subset of the program transformations offered by their heuristic counterparts, do not scale beyond small problems of up to 100 input instructions, and are not known to generate executable code.

The goal of this paper is to introduce a combinatorial approach to integrated register allocation and instruction scheduling that is practical. By *practical* we mean that the approach is:

Complete. It models the same set of program transformations as state-of-the-art compilers.
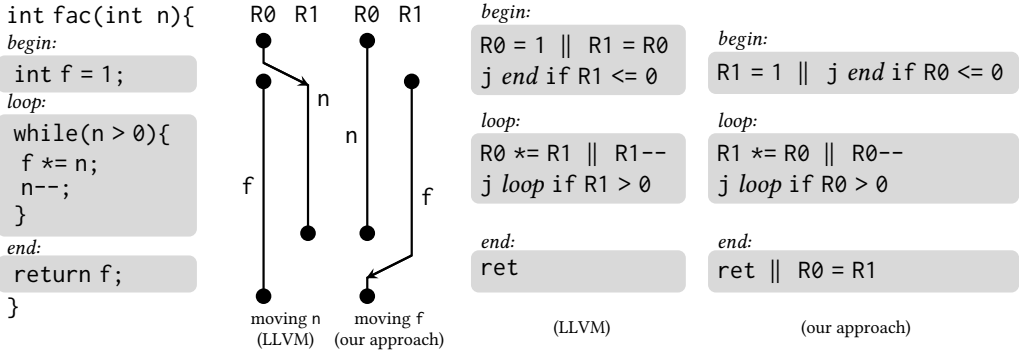**Scalable.** It scales to medium-sized problems of up to 1000 input instructions.
Executable. It generates executable code.

This combination of properties allows us, for the first time, to evaluate the potential benefit of combinatorial register allocation and instruction scheduling in practice: completeness enables a direct comparison with heuristic approaches; scalability puts most problems from typical benchmark suites within reach; and executability is a precondition to study the practical significance. Our evaluation shows that this potential benefit can be achieved, generating better code than state-of-the-art heuristic approaches for a variety of benchmarks, processors, and optimization goals.

Our approach has practical applications for both compiler users and developers, as corroborated by our research partner Ericsson [98]. Compiler users can apply it to trade compilation time for code quality beyond the usual compiler optimization levels. This is particularly attractive if longer compilation times are tolerable, such as for compilation of high-quality library releases or embedded systems software [53]. If compilation time is more limited, our approach might still be applied to the most frequently executed parts of a program. Compiler developers can apply the combinatorial approach as a baseline for assessment and development of heuristic approaches, exploiting the optimal, integrated nature of its solutions. Furthermore, comparing the generated code can reveal improvement opportunities in heuristic algorithms, even for production-quality compilers such as LLVM [18]. Another application is to generate high-quality code for processor-specific, irregular features. Such features can be naturally modeled and fully exploited by combinatorial approaches, while adapting heuristic approaches tends to involve a major effort [62]. The ease of modeling of our approach can also be exploited for rapid compiler prototyping for new processor revisions.

Example 1. Figure 1 illustrates our combinatorial approach in practice [17]. Figure 1a shows a C implementation of the factorial function. The function takes n as argument, initializes f to one, iterates multiplying and accumulating f with n in each iteration until n is zero, and returns f.

Register allocation for this function must assign n and f to different registers within the *loop* basic block, since their values would be otherwise clobbered (that is, mutually overwritten). Let us assume that the target processor is Hexagon, a multiple-issue digital signal processor ubiquitous in modern mobile platforms [30]. Its calling convention forces the assignment of n and f to the same register (R0) on entry and on return respectively [80]. These clobbering and calling convention constraints can only be satisfied by assigning one of the program variables $v \in \{n, f\}$ to some other

```
int fac(int n){
begin:
  int f = 1;
loop:
  while(n > 0){
   f *= n;
   n--;
  }
end:
  return f;
}
```

(a) Factorial in C.

moving n   moving f
(LLVM)   (our approach)

(b) Register assignment.

```
begin:
R0 = 1  ||  R1 = R0
j end if R1 <= 0

loop:
R0 *= R1  ||  R1--
j loop if R1 > 0

end:
ret
```

(LLVM)

```
begin:
R1 = 1  ||  j end if R0 <= 0

loop:
R1 *= R0  ||  R0--
j loop if R0 > 0

end:
ret  ||  R0 = R1
```

(our approach)

(c) Simplified assembly code.

Fig. 1. Motivating example: Hexagon assembly code generated by LLVM and our approach for factorial.
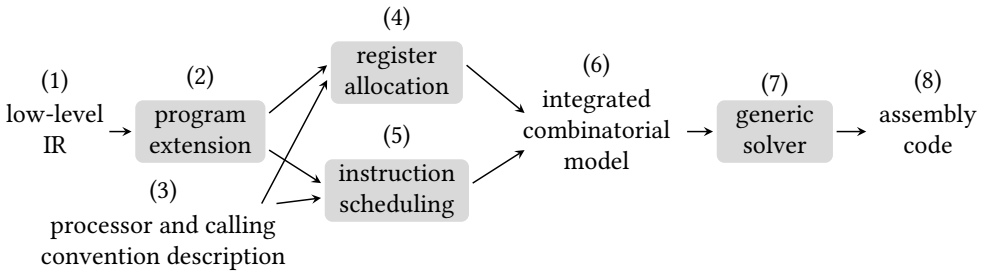

Fig. 2. Our approach to combinatorial register allocation and instruction scheduling.

register than R0 within *loop* and moving v to or from R0 outside of *loop* with a register-to-register move instruction.

Figure 1b shows how the state-of-the-art heuristic compiler LLVM [59] and our approach produce opposite assignments by moving either n or f. From an isolated register allocation perspective, both assignments incur the same cost: an additional move instruction at *begin* (LLVM) or *end* (our approach) is required. However, our integrated approach properly reflects that only the move of f can be parallelized (||) with another instruction (ret), yielding slightly faster assembly code (Figure 1c). This advanced optimization level requires reasoning about multiple aspects of global register allocation and instruction scheduling in integration.

Approach. Our approach is outlined in Figure 2. A low-level intermediate representation (IR) of a function with instructions from a specific processor is taken as input (1). The representation of the function is extended (2) to expose its structure and the multiple decisions involved in the problem. From the extended function and a description of the processor and calling convention (3), combinatorial models of register allocation (4) and instruction scheduling (5) are derived. A combinatorial model consists of variables representing the problem decisions, program and processor constraints over the variables, and an objective function to be optimized. Both models are then integrated into a single model (6) that precisely captures the interdependencies between the corresponding

problems. Solving the model with a generic solver (7) gives a register allocation and an instruction schedule used to generate executable assembly code (8).

A combinatorial model is of limited practical value unless complemented with suitable solving techniques, effective solving methods, and a robust implementation that gives confidence in the results. Our approach is implemented in *Unison* [23], a software tool that uses constraint programming [83] as a modern combinatorial optimization technique. Constraint programming is particularly capable of exploiting the structure underlying the register allocation and instruction scheduling problems. Unison applies general and problem-specific constraint solving methods with which medium-sized functions can be solved optimally. It also deals with the practicalities of generating executable code (such as calling convention and call stack management), delegating decisions that are interdependent with register allocation and instruction scheduling to the combinatorial model. Unison is a robust tool whose results can be used with confidence: it is open source, well-documented, systematically tested, used in industry, and integrated with the LLVM toolchain.

Contributions. This paper contributes the first combinatorial approach to register allocation and instruction scheduling that is *complete*, *scales* up to medium-sized problems, and generates *executable* code.

The combinatorial model is *complete* as it includes spilling (internalized into the model), register assignment and packing, coalescing, load-store optimization, live range splitting, rematerialization, and multi-allocation. This is enabled by a novel combination of abstractions that capture different aspects of register allocation together with a suitable program representation. In addition, the paper introduces model extensions for features such as stack frame elimination, latencies across basic blocks, operand forwarding, and two-address instructions.

The paper introduces solving methods that are crucial for *scalability*. Extensive experiments on MediaBench [61] and SPEC CPU2006 [90] functions for three different processor architectures (Hexagon, ARM, MIPS) show that, given a time limit of 15 minutes, our approach solves optimally functions of up to 946 instructions. Under this time limit, the percentage of functions solved optimally ranges from 44% to 79% across processors, and 90% of the functions on Hexagon are solved optimally or less than 10% away from the optimal solution.

The experiments confirm that there is a potential benefit to be gained by solving register allocation and instruction scheduling in integration. Our approach exploits this potential, delivering a mean estimated speedup over LLVM ranging from 1.1% to 10% and a mean code size reduction ranging from 1.3% to 3.8%, depending on the characteristics of the processor. For the first time, the speedup estimation is examined by *executing* MediaBench applications on Hexagon, the processor with highest estimated speedup in the experiments. The results show that the approach achieves significant speedup in practice (6.6% across functions and 5.6% across whole applications).

Plan of the paper. Section 2 covers the background on register allocation, instruction scheduling, and combinatorial optimization. Section 3 reviews related approaches.

Sections 4-9 introduce the combinatorial model and its associated program representation. They incrementally describe local register allocation (Section 4), global register allocation (Section 5), instruction scheduling (Section 6), the integrated model (Section 7), its objective function (Section 8), and additional program transformations (Section 9). Appendix A complements this incremental description with a summary of the parameters, variables, constraints, and objective function in the combinatorial model.

Section 10 outlines the solving methods employed by Unison. Section 11 presents the experimental evaluation, where Appendix B studies the accuracy of the speedup estimation and Appendix C describes the functions used in the evaluation. Section 12 concludes and proposes future work.

2 BACKGROUND

This section reviews the input program representation assumed in the paper, the register allocation and instruction scheduling problems, and constraint programming as the key technique for modeling and solving register allocation and instruction scheduling.

2.1 Input Program Representation

This paper assumes as input a function represented by its control-flow graph (CFG), with instructions for a specific processor already selected. Instructions *define* (assign) and *use* (access) program and compiler-generated variables. These variables are referred to as *temporaries* and are associated with a bit-width derived from their source data type.

Program points are locations between consecutive instructions. A temporary t defined by an instruction i is *live* at a program point if t holds a value that might be used later by another instruction j. In this situation, instruction j is said to be *dependent* on i. A temporary t is *live-in* (*live-out*) in a basic block b if t is live at the entry (exit) of b. The *live range* of a temporary t is the set of program points where t is live. Two temporaries *interfere* if their live ranges are not disjoint.

The paper assumes that all temporaries that are live-in into the function (such as function arguments) are defined by a *virtual entry instruction* at the entry basic block. Similarly, *virtual exit instructions* use temporaries that are live-out at the exit basic blocks of the function. *Virtual* (as opposed to *real*) instructions are pseudo-instructions introduced to support compilation and do not appear in the generated assembly code. Temporaries being live-in into the function and live-out of the function are pre-assigned according to the calling convention. A pre-assignment of a temporary t to a register r is denoted as $t \triangleright r$.

Example 2. Figure 3 shows the CFG from Example 1 with Hexagon instructions. For readability, Hexagon immediate transfer (`tfri`), conditional jump (`j if`), multiply (`mul`), subtract (`sub`), and jump to return address (`ret`) instructions are shown in simplified syntax. The body of a basic block is shown in dark gray while the boundaries containing entry and exit instructions are shown in light gray. For simplicity, *critical edges* (arcs from basic blocks with multiple successors to basic blocks with multiple predecessors) are preserved in the example CFG. In practice, such edges are often split by earlier compiler transformations.

The 32-bit temporaries n and f correspond to the `int` variables in Example 1. n is live-in and hence defined by the entry instruction in the *begin* basic block. f is live-out and hence used by the exit instruction in the *end* basic block. Hexagon's calling convention requires the pre-assignment n▷R0 on the function's entry, and f▷R0 on the function's exit. The live range of n starts with its definition by the entry instruction in the *begin* basic block and spans the entire *loop* basic block. The live range of f starts with its definition by `tfri` in the *begin* basic block and extends until the exit instruction in the *end* basic block. n and f interfere as their live ranges are not disjoint.

2.2 Register Allocation

Register allocation maps temporaries to either processor registers or memory. While processor registers have shorter access times, they are limited in number. This shortage of registers leads to a high *register pressure* for functions where many temporaries interfere. In the worst case, this may force the allocation (*spilling*) of some of the temporaries to memory. The spill of a temporary t is implemented by inserting code (often costly store and load instructions) to move t's value to and from memory. In its most basic form (called *spill-everywhere*), *spill code* is generated by inserting store instructions after each definition of t and load instructions before each use of t.

Register allocation applies a number of program transformations to reduce register pressure, spilling, or — if the latter is unavoidable — the overhead of the spill code:
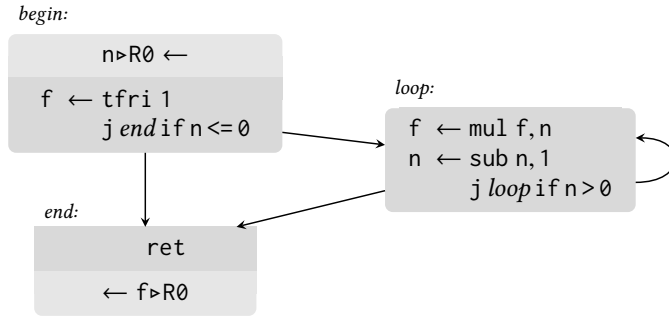
Fig. 3. CFG of the factorial function with Hexagon instructions.

Register assignment maps non-spilled temporaries to individual registers, reducing register pressure by reusing registers among non-interfering temporaries.

Live range splitting allocates temporaries to different locations in different parts of their live ranges.

Coalescing allocates a temporary's split live range to the same location to eliminate the corresponding copy instructions (dual of live range splitting).

Load-store optimization avoids reloading spilled values by reusing values loaded in previous parts of the spill code.

Multi-allocation allocates temporaries simultaneously to registers as well as memory, reducing the overhead of spilling in certain scenarios [31].

Register packing assigns temporaries of small bit-widths to different parts of larger-width registers (for processors supporting such assignments) to improve register utilization.

Rematerialization recomputes values at their use points rather than spilling them when the recomputation instructions are deemed less costly than the spill code.

Scope. Register allocation can be solved locally or globally. *Local* register allocation handles single basic blocks, spilling all temporaries that are live across basic block boundaries. *Global* register allocation handles entire functions, potentially improving code quality by keeping temporaries in registers across basic block boundaries.

Calling convention management. Register allocation is responsible to enforce the pre-assignment of arguments and return values to certain registers and the preservation of *callee-saved* registers across function calls as dictated by the calling convention. A common approach to handle callee-saved registers is to hold their values in temporaries that are live through the entire function and that can be spilled at the function's entry and exit. This approach can be refined by moving the spill code closer to the program points where the callee-saved registers are needed (*shrink-wrapping* [27]).

SSA-based register allocation. Static single assignment (SSA) [34] is a program form in which temporaries are defined exactly once. This form is useful for register allocation as it allows register assignment to be solved optimally in polynomial time [48]. SSA places ϕ-functions of the form $t_n \leftarrow \phi(t_1, t_2, \ldots, t_{n-1})$ at the entry of basic blocks with multiple incoming arcs to distinguish which definition among $\{t_1, t_2, \ldots, t_{n-1}\}$ reaches t_n depending on the program execution. Temporaries related transitively by ϕ-functions are called ϕ-*congruent* [92]. This paper uses a generalized form of SSA for global register allocation, as Section 5.1 explains.

2.3 Instruction Scheduling

Instruction scheduling assigns issue cycles to program instructions. Valid instruction schedules must satisfy instruction dependencies and constraints imposed by limited processor resources.

Latency. Dependencies between instructions are often associated with a latency indicating the minimum number of cycles that must elapse between the issue of the depending instructions. Variable latencies (such as those arising from cache memory accesses) are typically handled by assuming the best case and relying on the processor to stall the execution otherwise [47].

Resources. The organization of hardware resources varies heavily among different processors and has a profound impact on the complexity of instruction scheduling. This paper assumes a resource model where each resource s has a capacity $cap(s)$ and each instruction i consumes $con(i, s)$ units of each resource s during $dur(i, s)$ cycles. This model is sufficiently general to capture the resources of the processors studied in this paper, including Very Long Instruction Word (VLIW) processors such as Hexagon that can issue multiple instructions in each cycle. These processors can be modeled by an additional resource with capacity equal to the processor's issue width.

Order. Non-integrated compilers perform instruction scheduling before or after register allocation. Pre-register-allocation scheduling typically seeks a schedule that reduces register pressure during register allocation, while post-register-allocation scheduling often aims at minimizing the duration of the schedule. The latter is particularly relevant for *in-order* processors which issue instructions according to the schedule produced by the compiler, although *out-of-order* processors can also benefit from compile-time instruction scheduling [47].

Scope. Instruction scheduling can be solved at different program scopes. This paper is concerned with *local* instruction scheduling, where the instructions from each basic block are scheduled independently. Larger scopes to which instruction scheduling can be applied include *superblocks* (consecutive basic blocks with multiple exit points but a single entry point) and *loops* (where the instructions of multiple iterations are scheduled simultaneously in a pipelined fashion).

2.4 Constraint Programming

Combinatorial optimization is a collection of techniques to solve computationally hard combinatorial optimization problems such as register allocation and instruction scheduling. Examples of these techniques are constraint programming (CP) [83], integer programming (IP) [75], Boolean satisfiability (SAT) [12], partitioned Boolean quadratic programming (PBQP) [85], and dynamic programming (DP) [33]. Their key property is *completeness*: they automatically explore the full solution space and, given enough time, guarantee to find the optimal solution if there is one.

Combinatorial optimization techniques capture the problems to be solved as combinatorial models. A combinatorial model consists of *variables* capturing problem decisions, *constraints* expressing relations over the variables that must be satisfied by a solution, and an *objective function* to be minimized (or maximized) expressed in terms of the variables. A *solution* to the model is an assignment of values to the variables that satisfies all the constraints, and an *optimal solution* minimizes the value of the objective function. In this paper variables are written in bold (e.g. $\mathbf{x}$), and indexable variables are written as $\mathbf{x}(i)$.

Combinatorial optimization techniques differ significantly in the level of abstraction of their models, underlying solving methods, and problem classes for which they are particularly well-suited. This paper uses CP as a technique that is particularly suitable for handling register allocation and instruction scheduling problems. In CP, variables usually range over finite subsets of the integers or Booleans, and constraints and objective function are expressed by general relations. The purpose

of this section is to provide enough information so that the models developed in Sections 4-9 are understandable. Some additional modeling and solving methods are presented in Section 10.

Constraint solvers proceed by interleaving *constraint propagation* and *branch-and-bound search*. Constraint propagation discards value assignments that cannot be part of a solution to reduce the search space. Constraint propagation is applied iteratively until no more value assignments can be discarded [11]. If several values can still be assigned to a variable, search is used to decompose the problem into alternative subproblems on which propagation and search are repeated. Solutions found during solving are exploited in a branch-and-bound fashion to further reduce the search space [93]: after a solution is found, constraints are added such that the next solution must be better according to the objective function.

As is common in combinatorial optimization, constraint solvers are usually run with a time limit as it can be prohibitive to find the optimal solution to large problems. Constraint solvers exhibit *anytime behavior* in that they can often deliver suboptimal solutions if they time out. Even when no solution is delivered, the solvers always provide the *optimality gap* (hereafter just *gap*), an upper bound on the distance to the optimal solution. The gap can be used as an estimation of the computational effort to obtain the optimal solution, as a certificate of the quality of suboptimal solutions, or as a criterion for solver termination.

Global constraints. A key feature of CP is the use of *global constraints* that capture recurring modeling substructures involving multiple variables. Besides being convenient for modeling, global constraints are essential for efficient solving since they are implemented by dedicated propagation algorithms that further reduce the search space [95]. The model introduced in this paper (Sections 4-9) uses the following three global constraints:

- cumulative $(\{\langle \mathbf{s}(i), \mathbf{d}(i), \mathbf{c}(i) \rangle : i \in (1, n)\}, b)$ ensures that a set of n tasks does not exceed a given resource capacity b, where each task i is defined by its start time $\mathbf{s}(i)$, duration $\mathbf{d}(i)$, and resource units consumed $\mathbf{c}(i)$ [1]:

$$\sum_{i \in (1,n):\mathbf{s}(i) \leq k \wedge \mathbf{s}(i)+\mathbf{d}(i) > k} \mathbf{c}(i) \leq b \quad \forall k \in \mathbb{Z}. \tag{1}$$

- no-overlap $(\{\langle \mathbf{l}(i), \mathbf{r}(i), \mathbf{t}(i), \mathbf{b}(i) \rangle : i \in (1, n)\})$ (also known as *diffn*) ensures that a set of n rectangles does not overlap, where each rectangle i is defined by its left $\mathbf{l}(i)$, right $\mathbf{r}(i)$, top $\mathbf{t}(i)$, and bottom $\mathbf{b}(i)$ coordinates [10]:

$$\mathbf{r}(i) \leq \mathbf{l}(j) \vee \mathbf{l}(i) \geq \mathbf{r}(j) \vee \mathbf{b}(i) \leq \mathbf{t}(j) \vee \mathbf{t}(i) \geq \mathbf{b}(j) \quad \forall i, j \in (1, n) : i \neq j. \tag{2}$$

- element $(\mathbf{x}, a, \mathbf{y})$ generalizes array access to variables by ensuring that the variable $\mathbf{y}$ is equal to the $\mathbf{x}^{\text{th}}$ element of an array a of integers or integer variables [94]:

$$a[\mathbf{x}] = \mathbf{y}. \tag{3}$$

As a compromise between readability and precision, the rest of the paper expresses this constraint using the array notation. This can be extended to multi-dimensional arrays provided that only a single integer variable is used for indexing. Constraints using lookup functions, such as $f(\mathbf{x}) = \mathbf{y}$, can be expressed as $f[\mathbf{x}] = \mathbf{y}$ by converting f into an array. Using this constraint, we can also implement the constraint

$$\mathbf{y} \in a[\mathbf{x}] \tag{4}$$

by introducing, for each such constraint, a new array b of integer variables, where each integer variable $b[i]$ has a domain equal to $a[i]$, and enforcing $b[\mathbf{x}] = \mathbf{y}$.

Table 1. Combinatorial register allocation approaches ordered chronologically: technique, scope, spilling (**SP**), register assignment (**RA**), coalescing (**CO**), load-store optimization (**LO**), register packing (**RP**), live range splitting (**LS**), rematerialization (**RM**), multi-allocation (**MA**), size of largest problem solved optimally (**SZ**) in input instructions, and whether it is demonstrated to generate executable code (**EX**).

approach	technique	scope	SP	RA	CO	LO	RP	LS	RM	MA	SZ	EX
Goodwin *et al.* [46]	IP	global	✓	✓	✓	✓	✓	✓	✓	✓	∼ 2000	✓
Appel *et al.* [3]	IP	global	✓			✓		✓			∼ 2000	✓
Scholz *et al.* [85]	PBQP	global	✓	✓	✓		✓		✓		∼ 180	✓
Nandivada *et al.* [73]	IP	global	✓	✓		✓		✓			?	✓
Koes *et al.* [57]	IP	global	✓	✓		✓		✓	✓		?	✓
Barik *et al.* [8]	IP	global	✓	✓		✓	✓	✓	✓		302	
Ebner *et al.* [35]	IP	global	✓			✓		✓			?	✓
Falk *et al.* [38]	IP	global	✓	✓		✓		✓		✓	∼ 1000	✓
Colombet *et al.* [31]	IP	global	✓			✓		✓	✓	✓	?	✓

3 RELATED APPROACHES

This section provides an overview of combinatorial approaches to register allocation and instruction scheduling. The overview is complemented by more specific discussions in the rest of the paper. A more comprehensive review is provided by Castañeda Lozano and Schulte [24].

Combinatorial register allocation. Combinatorial approaches to register allocation in isolation (Table 1) have been proposed that satisfy all properties required to be *practical*: they model most or all of the standard program transformations (*completeness*, columns **SP-MA**), scale to medium-sized problems (*scalability*, column **SZ**), and generate executable code (*executability*, column **EX**). Furthermore, their ability to accommodate specific architectural features and alternative optimization objectives has been demonstrated in numerous studies [8, 38, 72, 73]. A particular focus has been to study the trade-off between solution quality and scalability. Numerous approaches [3, 31, 35] advocate solving spilling first (including relevant aspects of coalescing in the case of Colombet *et al.* [31]), followed by register assignment and coalescing. This arrangement can improve scalability with virtually no performance loss for single-issue and out-of-order processors, but is less suitable when register assignment and coalescing have high impact on code quality, such as in code size optimization [58] or in speed optimization for VLIW processors [31].

Combinatorial instruction scheduling. A large number of combinatorial instruction scheduling approaches have been proposed, where the underlying resource-constrained project scheduling problem has already been solved with IP in the early 1960s [13, 68, 96]. *Practical* approaches have been proposed for both local [86] and global [7, 103] instruction scheduling. Other approaches that scale to medium-sized problems but do not demonstrate executable code have been proposed for local [67, 100] and superblock [66] instruction scheduling. A detailed review of combinatorial instruction scheduling is provided by Castañeda Lozano and Schulte [24].

Combinatorial register allocation and instruction scheduling. Integrated combinatorial approaches (Table 2) capture the interdependencies between register allocation and instruction scheduling in a single combinatorial model. They typically are less scalable but can deliver better solutions than isolated approaches. A particular focus has been on VLIW processors, for which the interdependencies between both problems are strong [55]. As Table 2 shows, our integrated approach is the first that matches the practicality of isolated register allocation approaches. This concerns the program transformations modeled (*completeness*, columns **SP-MA**), the *scalability* (column **SZ**),

Table 2. Combinatorial register allocation and instruction scheduling approaches ordered chronologically: technique, scope, spilling (**SP**), register assignment (**RA**), coalescing (**CO**), load-store optimization (**LO**), register packing (**RP**), live range splitting (**LS**), rematerialization (**RM**), multi-allocation (**MA**), instruction selection (**SE**), size of largest problem solved optimally (**SZ**) in input instructions, and whether it is demonstrated to generate executable code (**EX**).

approach	technique	scope	SP	RA	CO	LO	RP	LS	RM	MA	SE	SZ	EX
Wilson *et al.* [101]	IP	global	✓	✓	✓			✓			✓	30	
Gebotys [41]	IP	local	✓	✓		✓		✓			✓	108	
Chang *et al.* [26]	IP	local	✓			✓						∼ 10	
Bashford *et al.* [9]	CP	local	✓	✓	✓	✓		✓			✓	23	
Kästner [54]	IP	superblock	✓									39	
Kessler *et al.* [56]	DP	local						✓		✓	✓	42	
Nagarakatte *et al.* [71]	IP	loop	✓	✓		✓		✓		✓		?	
Eriksson *et al.* [37]	IP	loop	✓			✓		✓		✓	✓	100	
(this paper)	**CP**	**global**	✓	✓	✓	✓	✓	✓	✓	✓		**946**	✓

and the *executability* (column **EX**). While earlier integrated approaches might be able to generate executable code, this fact is not demonstrated in the respective publications, which precludes an evaluation of their practical significance. Moreover, only the approach by Wilson *et al.* [101] models register allocation at the global scope.

A particularly challenging problem not addressed by this paper is to incorporate instruction selection (column **SE**). The complexity of instruction selection and its interdependencies with register allocation and instruction scheduling are well-understood [50]. However, the potential benefits and scalability challenges of incorporating instruction selection into our integrated approach have not yet been explored.

Optimization techniques. IP is the most widely used optimization technique with the exception of Scholz and Eckstein [85] using PBQP, Bashford and Leupers [9] using CP, and Kessler and Bednarski [56] using DP. For combinatorial instruction scheduling in isolation a broader set of techniques has been applied, including the use of special-purpose branch-and-bound techniques [24].

4 LOCAL REGISTER ALLOCATION

This section introduces the combinatorial model for local register allocation and its underlying program transformations. The model is introduced step-by-step: starting from a definition of the input programs (Section 4.1) and register assignment (Section 4.2), the model is extended with alternative instructions (Section 4.3); spilling, live range splitting, and coalescing (Section 4.4); and rematerialization (Section 4.5). Section 4.6 briefly summarizes the main contributions in the model and program representation.

In the remainder of the paper, each variable added to the model is numbered by (Vn) where n is a number. Likewise, constraints are numbered by (Cn), where n is either a number or a period-separated number pair in case the constraint has been refined or extended.

4.1 Program Representation

The model for local register allocation is parameterized with respect to the input program and processor. This section defines the basic program parameters that describe an input program. Processor parameters and the remaining program parameters are introduced as needed.

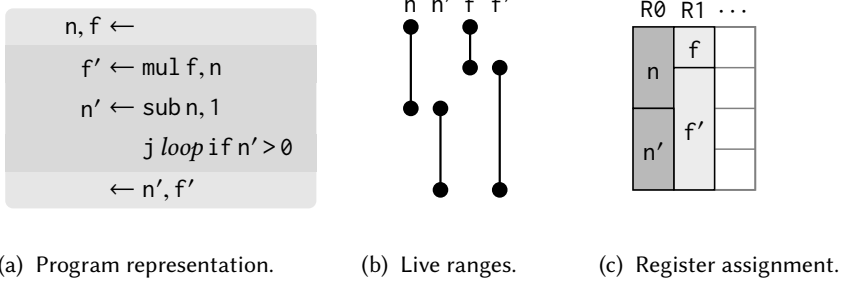(a) Program representation.　　(b) Live ranges.　　(c) Register assignment.

Fig. 4. Local register assignment for the *loop* basic block from Example 2.

The input program consists of a sequence of operations in a single basic block. A key idea in the program representation is that it distinguishes between *operations* and *instructions* as well as between *operands* and *temporaries* to make the model more powerful. Operations are seen as abstract instructions (such as addition) that are implemented by specific processor instructions (such as add). Distinguishing between operations and instructions enables supporting alternative instructions, see Section 4.3. Operations contain *operands* that denote particular use and definition points of temporaries. Distinguishing between operands and temporaries enables register allocation transformations beyond register assignment and packing, as Sections 4.4 and 4.5 show.

Live-in (live-out) temporaries in a basic block are defined (used) by an entry (exit) operation, corresponding to the virtual instructions discussed in Section 2.1. This ensures that the live range of a temporary can be derived simply from the operations defining and using the temporary. An operation using or defining a temporary t is called a *user* respectively the *definer* of t.

Programs in this representation are described by a set of operations O, operands P, and temporaries T. An operation implemented by instruction i that uses temporaries t_1 to t_n and defines temporaries t_{n+1} to t_m through its corresponding operands is represented as

$$p_{n+1} : t_{n+1}, \ldots, p_m : t_m \leftarrow i \; p_1 : t_1, \ldots, p_n : t_n.$$

The temporary used or defined by each operand p is denoted as $temp(p)$. For simplicity, operand identifiers $p_1, p_2, \ldots, p_m$ are omitted if possible.

As the input program is assumed to be in SSA, each temporary is defined exactly once. The program point immediately after the single definition of a temporary t is denoted as $start(t)$, whereas the program point immediately before the last use of t is denoted as $end(t)$. In our setup, the live range of a temporary t is indeed a range (or interval) $(start(t), end(t))$, and can be enforced straightforwardly in single basic blocks by local value numbering [32]. Live ranges being simple intervals is essential for modeling register assignment as Section 4.2 shows.

Example 3. Figure 4a illustrates the input representation of the *loop* basic block from Example 2 for local register allocation. Operand identifiers are omitted for simplicity. The live-in temporaries n and f are defined by an entry operation. The redefinitions of f (by mul) and n (by sub) are renamed as f′ and n′ to enforce SSA. Single definitions result in interval live ranges as shown in Figure 4b. The live-out temporaries n′ and f′ are used by an exit operation.

4.2 Register Assignment

This section introduces a simple model for register assignment, where temporaries are mapped to individual registers. The mapping is modeled by a variable **reg**(t) for each temporary t giving its assigned register from a register set R. This paper uses symbolic domains for clarity; the actual

model maps symbolic values to integer values:

$$\mathbf{reg}(t) \in R \quad \forall t \in T. \tag{V1}$$

Register assignment is constrained by interference among temporaries. For example, n and f in Figure 4a interfere and therefore cannot be assigned to the same register. Register assignment and interference are captured by a simple geometric interpretation that can be modeled by global constraints. This interpretation, based on Pereira and Palsberg's *puzzle solving* approach [77], projects the register assignment of a basic block onto a rectangular area, where the horizontal dimension corresponds to an ordering of R (called *register array*) and the vertical dimension corresponds to program points. Each temporary t yields a rectangle of $width(t) = 1$, where the top and bottom borders correspond to the $start(t)$ and $end(t)$ program points of t's live range and the left border corresponds to the register $\mathbf{reg}(t)$ to which t is assigned. The direct mapping from temporaries to rectangles is possible due to the interval shape of the temporary live ranges. In this interpretation, the rectangles of interfering temporaries overlap vertically. A single *no-overlap* constraint in the model ensures that interfering temporaries are assigned to different registers:

$$\text{no-overlap} \left(\{ \langle \mathbf{reg}(t), \mathbf{reg}(t) + width(t), start(t), end(t) \rangle : t \in T \} \right) . \tag{C1}$$

Example 4. Figure 4c shows the geometric interpretation of a register assignment for the *loop* basic block where n and n′ are assigned to R0 and f and f′ are assigned to R1. Assuming the program points are numbered from one to five, the corresponding instantiation of constraint C1 becomes no-overlap $(\{\langle \mathbf{reg}(f), \mathbf{reg}(f) + 1, 1, 2 \rangle, \langle \mathbf{reg}(f'), \mathbf{reg}(f') + 1, 2, 5 \rangle, \langle \mathbf{reg}(n), \mathbf{reg}(n) + 1, 1, 3 \rangle, \langle \mathbf{reg}(n'), \mathbf{reg}(n') + 1, 3, 5 \rangle\})$.

Register packing. Register packing assigns temporaries of small bit-widths to different parts of larger-width registers to improve register utilization. This program transformation can be easily accommodated within the geometric representation of local register assignment. The registers in R are decomposed into their underlying register *atoms* (minimum register parts addressable by instructions), the function $width(t)$ is extended to give the number of adjacent atoms that temporary t occupies, and the variable $\mathbf{reg}(t)$ is reinterpreted as the leftmost atom to which t is assigned. This extension generalizes the geometric representation as a rectangle packing problem with rectangles of different widths and heights, which is precisely captured by constraint C1 as is.
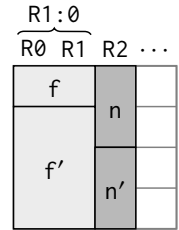

Fig. 5. Reg. packing.

Example 5. Figure 5 illustrates register packing for a version of the *loop* basic block in Example 3 where f and f′ have a width of 64 rather than 32 bits. Hexagon's register atoms (R0, R1, …) have a width of 32 bits and can be combined into 64-bit registers (R1:0, R3:2, …), hence $width(f) = width(f') = 2$. In the example register packing, f and f′ are assigned to the combined register R1:0. This is modeled by the assignment of the variables $\mathbf{reg}(f)$ and $\mathbf{reg}(f')$ to R0, the leftmost atom of R1:0. n′ and n′ are assigned to R2.

Pre-assignments. As discussed in Section 2.1, an operand p might be pre-assigned to a certain register r (given by $p \triangleright r$). Temporaries used or defined by such operands are pre-assigned to the corresponding registers:

$$\mathbf{reg}(temp(p)) = r \quad \forall p \in P : p \triangleright r. \tag{C2}$$

Register classes. Register files tend to be irregular in that different instructions are allowed to access different subsets of the available registers. A subset of the registers available to a particular instruction or group of instructions is called a *register class*. The processor parameter $class(p)$ gives

the register class of operand p. The registers to which a temporary can be assigned are determined by the operands that define and use the temporary:

$$\mathbf{reg}(temp(p)) \in class(p) \quad \forall p \in P. \tag{C3}$$

Section 4.4 below exploits register classes for spilling, where memory is treated as an additional register class, albeit with an unlimited number of registers.

Contributions. This section contributes a novel use of Pereira and Palsberg's register assignment and packing approach [77], by incorporating it into a combinatorial model. This is possible as constraint programming readily provides efficient solving methods for rectangle packing in the form of dedicated propagation algorithms for the *no-overlap* constraint. Integer programming, an alternative technique that is popular for register allocation (see Table 1), has difficulties in dealing with the disjunctive nature of rectangle packing [63].

4.3 Alternative Instructions

The model is extended by *alternative instructions* that can be selected to implement operations relevant for register allocation. For example, ARM's Thumb-2 instruction set extension allows 16- and 32-bit instructions to be freely mixed, where the 16-bit instructions reduce code size but can only access a subset of the registers. The selection of 16- or 32-bit instructions is interdependent with register allocation as it has a significant effect on register pressure [36]. Alternative instructions are also central to model spilling, live range splitting, and coalescing as Section 4.4 explains.

The model supports alternative instructions as follows. The set of instructions that can implement an operation o is given by the parameter $instrs(o)$, and the function $class(p, i)$ is redefined to give the register class of operand p if implemented by instruction i. An operation that can be implemented by alternative instructions $i_1, \ldots, i_n$ is represented as $\cdots \leftarrow \{i_1, \ldots, i_n\} \cdots$.

A variable $\mathbf{ins}(o)$ for each operation o gives the instruction that implements o:

$$\mathbf{ins}(o) \in instrs(o) \quad \forall o \in O. \tag{V2}$$

The register class of each operand p is linked to the instruction that implements p's operation in constraint C3 (changes to constraint C3 are highlighted in gray):

$$\mathbf{reg}(temp(p)) \in class\ [p, \mathbf{ins}(operation(p))] \quad \forall p \in P \tag{C3.1}$$

where the parameter $operation(p)$ gives the operation of operand p. Multi-dimensional array access and set membership can be expressed with *element* constraints as described in Section 2.4.

4.4 Spilling, Live Range Splitting, and Coalescing

Spilling (allocate temporaries into memory, including load-store optimization and multi-allocation), live range splitting (allocate temporaries to different locations along their live ranges), and coalescing (assign copied temporaries to the same register) are supported by a single extension of the program representation and the model. The extension is based on the notion of optional copy operations.

A copy operation $t_d \leftarrow \{i_1, \ldots, i_n\}\ t_s$ defines a destination temporary t_d with the value of a source temporary t_s using a copy instruction among a set of alternatives $\{i_1, \ldots, i_n\}$. Another operation using t_s in the original program might after the introduction of the copy use t_d as an alternative, because t_s and t_d hold the same value as they are *copy-related*. If some operation uses t_d, the temporary is considered *live* and its definer copy operation *active*. Otherwise, t_d is considered *dead* and its definer copy operation *inactive*, effectively coalescing t_d into t_s. Inactive operations and dead temporaries do not affect the constraints in which they occur. Copy operations can be: implemented by store and load instructions to support spilling; implemented by register-to-register move instructions to support live range splitting; or made inactive to support coalescing.

$$
\begin{array}{ll}
t \leftarrow \cdots & t \leftarrow \cdots \\
 & t_1 \leftarrow \{\texttt{store}, \texttt{move}\}\ t \\
\vdots & \vdots \\
 & t_2 \leftarrow \{\texttt{load}, \texttt{move}\}\ \{t, t_1\} \\
\cdots \leftarrow t & \cdots \leftarrow p : \{t, t_1, t_2, \ldots, t_n\} \\
\vdots & \vdots \\
 & t_n \leftarrow \{\texttt{load}, \texttt{move}\}\ \{t, t_1\} \\
\cdots \leftarrow t & \cdots \leftarrow \{t, t_1, t_2, \ldots, t_n\}
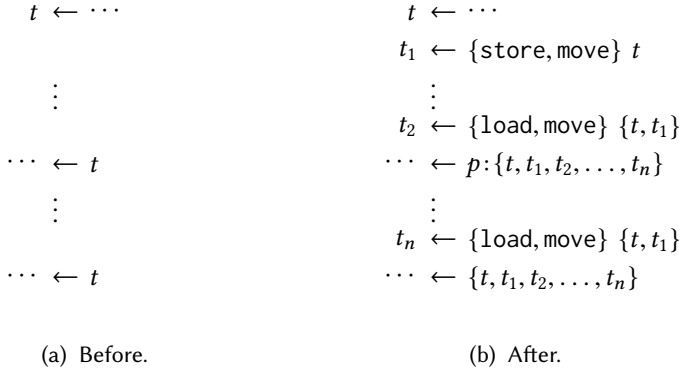\end{array}
$$

(a) Before.   (b) After.

Fig. 6. Copy extension.

To handle store and load instructions and register-to-register move instructions uniformly, the register array is extended with a *memory* register class containing an unbound number of registers $\{\texttt{M0}, \texttt{M1}, \ldots\}$ corresponding to memory locations in the function's stack frame. Unifying registers and memory yields a simple and expressive model that internalizes spilling by reducing register allocation to register assignment, where the register assigned to a temporary t implies t's allocation.

Copy extension. The number, type, and relation among the copy operations with which a program is extended to support spilling and live range splitting depends on the targeted processor architecture. This paper assumes load-store processors where values can be copied between registers and memory by single instructions. However, the model supports more complex architectures such as clustered VLIW processors by introducing additional copy operations.

For load-store processor architectures, a program is extended by visiting each temporary t once and introducing a *store-move* copy operation with instructions store and move at the program point immediately after the the definition of t and a *load-move* copy operation with instructions load and move at the program point immediately before each use of t. The store-move can only use t, each load-move might use t or the destination temporary of the store-move, and each original user of t might use t or any of the introduced temporaries. Figure 6 illustrates the transformation.

Note that copy extension suggests the use of some alternative temporaries at points where they are not yet defined, for example t_n used by operand p in Figure 6. Such uses are invalid in isolated register allocation where operations are totally ordered. However, they become relevant in the integration with instruction scheduling (see Section 7), where operations (including copy operations) can be rearranged.

Model extension. The new decisions that follow from the introduction of copy operations are captured by three classes of variables. A variable **temp**(p) for each operand p gives the temporary that is used or defined by operand p among a set of copy-related alternatives $temps(p)$:

$$\textbf{temp}(p) \in temps(p) \quad \forall p \in P; \tag{V3}$$

a Boolean variable **live**(t) indicates whether temporary t is live:

$$\textbf{live}(t) \in \mathbb{B} \quad \forall t \in T; \tag{V4}$$

and a Boolean variable **active**(o) indicates whether operation o is active:

$$\textbf{active}(o) \in \mathbb{B} \quad \forall o \in O. \tag{V5}$$

As is common in constraint programming, we define the set $\mathbb{B}$ as $\{0, 1\}$ and abbreviate, for example, **active**$(o) = 1$ as **active**(o) and **active**$(o) = 0$ as $\neg$**active**(o).

For uniformity, **temp**(p) is defined for both use and define operands even though no alternative is available for the latter. Similarly, **active**(o) is defined for both copy and non-copy operations even though the latter must always be active:

$$\textbf{active}(o) \quad \forall o \in O : \neg copy(o) . \tag{C4}$$

For a live temporary t, the definer of t and at least one user of t must be active:

$$\begin{aligned} \textbf{live}(t) &\iff \textbf{active}(operation(definer(t))) \\ &\iff \exists p \in users(t) : \textbf{active}(operation(p)) \wedge \textbf{temp}(p) = t \quad \forall t \in T, \end{aligned} \tag{C5}$$

where $definer(t)$ and $users(t)$ are the operand(s) that might define and use a temporary t, and $operation(p)$ is the operation of operand p.

With the introduction of alternative temporaries, the temporary used or defined by operands involved in pre-assignments becomes variable:

$$\textbf{reg} \left[\textbf{temp}(p) \right] = r \quad \forall p \in P : p \triangleright r \tag{C2.1}$$

Additionally, the register class constraint only needs to consider active operations:

$$\textbf{reg} \left[\textbf{temp}(p) \right] \in class[p, \textbf{ins}(operation(p))] \quad \forall p \in P : \textbf{active}(operation(p)) . \tag{C3.2}$$

Example 6. Figure 7 illustrates the application of copy extension to support different implementations of spilling for the running example. Figure 7a shows the result of applying copy extension to n using Hexagon's store (stw), load (ldw), and register-to-register move (tfr) instructions.

Figure 7b illustrates the spill of n on the register array extended with memory registers. n is stored directly after its definition (copied to n_1 which is assigned to memory register M0) and loaded before each use as n_2 and n_3 in a spill-everywhere fashion. This illustrates how the approach supports multi-allocation: immediately before mul, n_1 and n_2 are live simultaneously in memory (M0) and in a processor register (R0), simulating multiple allocations of their original temporary n.

Figure 7c illustrates the spill of n where load-store optimization is applied, rendering the second load-move inactive. Load-store optimization is only possible due to the availability of n_2 as an alternative use for sub.

Live range splitting is supported similarly to spilling. For example, the live range of n can be split by implementing the copy operations that are active in Figure 7c with tfr instructions, and letting constraint C3.2 force the assignment of n_1 and n_2 to processor registers.

Discussion. The local register allocation model introduced in this section is the first to exploit copy operations, copy extension, and memory registers as a uniform abstraction to capture spilling, live range splitting, coalescing, and rematerialization (to be explained in Section 4.5). Only the related approaches of Wilson *et al.* [101, 102] and Chang *et al.* [26] extend the program representation with explicit copy operations and treat them exactly as original program operations. However, they do not model memory and processor registers uniformly, which requires special treatment of spilling and hence results in a more complicated model.

Alternative temporaries are related to Colombet *et al.*'s *local equivalence classes* [31] in their definition and targeted program transformations, and to Wilson *et al.*'s *alternative implementations* [101, 102] in their way of extending the original program representation with implementation choices to be considered by a solver.

Copy extension for load-store processors (Figure 6) allows a temporary's live range to be split at each definition and use point. Introducing even more splitting choices at other program points

(a) Copy extension of n.   (b) Spill-everywhere.   (c) Load-store optimization.

Fig. 7. Copy extension and spilling for n in the *loop* basic block.

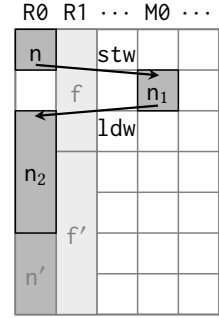might enable better code. The number of these choices and their distribution among program points differs considerably among combinatorial register allocation approaches. This ranges from no live range splitting (as in Scholz and Eckstein [85]) to *split-everywhere* at every program point (as introduced by Appel and George [3]). The potential loss in code quality by bounding live range splitting to definition and use points is not yet established. In any case, the potential loss is mitigated as the model is expanded to global register allocation in Section 5 (where temporaries can be split at the boundaries of each basic block) and integrated with instruction scheduling in Section 7 (where the copy operations performing the splits can be rearranged).

4.5 Rematerialization

Rematerialization recomputes values at their use points rather than spilling them if the recomputation instructions are sufficiently cheap. A common case is rematerialization of *never-killed* values [25] that can be recomputed by a single instruction from operands that are always available. Typical examples are numeric constants and variable addresses in the stack frame. This paper captures rematerialization of never-killed values by a simple adjustment of copy extension and the register array and does not require any changes to the actual model. The level of rematerialization is similar to that of related approaches in Table 1.

Rematerializable temporaries are identified using the data-flow analysis of Briggs *et al.* [14]. The register array is extended with a *rematerialization* register class containing an unbounded number of registers (RM0, RM1, ...). Rematerialization of a temporary t is modeled as if t were defined in a rematerialization register by a virtual instruction remat and loaded into a processor register by t's actual definer. This is achieved by adjusting the copy extension of each rematerializable temporary t to include the virtual instruction remat as an alternative to t's original definer instruction i, and include i as an alternative to the instructions of each load-move. Figure 8 illustrates this adjustment.

Modeling rematerialization as an alternative load-move instruction is inspired by Colombet *et al.* [31]. Unlike Colombet *et al.*, we use copy operations and extend the register array to avoid additional variables and constraints.

4.6 Summary

This section has introduced a program representation and model that enable all transformations offered by local, state-of-the-art register allocation using few variables and constraints. The compact,

$$t \leftarrow i \cdots \qquad\qquad\qquad t \leftarrow \{i, \mathsf{remat}\} \cdots$$
$$t_1 \leftarrow \{\mathsf{store}, \mathsf{move}\}\, t$$
$$\vdots \qquad\qquad\qquad\qquad\qquad \vdots$$
$$t_2 \leftarrow \{\mathsf{load}, \mathsf{move}, i\}\, \{t, t_1\}$$
$$\cdots \leftarrow t \qquad\qquad\qquad \cdots \leftarrow \{t, t_1, t_2, \ldots, t_n\}$$
$$\vdots \qquad\qquad\qquad\qquad\qquad \vdots$$
$$t_n \leftarrow \{\mathsf{load}, \mathsf{move}, i\}\, \{t, t_1\}$$
$$\cdots \leftarrow t \qquad\qquad\qquad \cdots \leftarrow \{t, t_1, t_2, \ldots, t_n\}$$
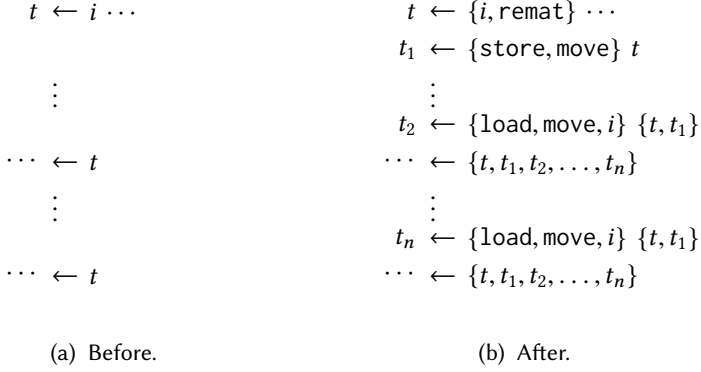
(a) Before. (b) After.

Fig. 8. Adjusted copy extension for a rematerializable temporary.

yet expressive model is based on two significant contributions to the state of the art in combinatorial register allocation. First, the model incorporates Pereira and Palsberg's register assignment and packing approach [77] into a combinatorial setup. Second, it introduces copy operations, copy extension, and memory registers as a uniform abstraction to model spilling, live range splitting, coalescing, and rematerialization.

5 GLOBAL REGISTER ALLOCATION

This section extends the model from local to global register allocation for entire functions.

5.1 Program Representation

Global register allocation represents an input function as a set of basic blocks B where each basic block is defined as in Section 4.1. For a basic block b, O_b is the set of operations in b and T_b is the set of temporaries in b. The disjoint unions of these sets are denoted as O and T.

The input function is assumed to be in *linear static single assignment* (*LSSA*) form. LSSA generalizes SSA by enforcing that each temporary is defined once and only used within the basic block where it is defined [2, 5]. LSSA construction decomposes each temporary t with a live range spanning multiple blocks $b_1, b_2, \ldots, b_n$ into a set of *congruent* temporaries $t_{b_1} \equiv t_{b_2} \equiv \cdots \equiv t_{b_n}$, where each temporary is local to its respective basic block. Hence the congruences denote that two or more temporaries originate from the same temporary and must therefore be assigned the same register. Live-in (live-out) temporaries resulting from LSSA construction are defined (used) by entry (exit) operations. The conventional form [92] of LSSA is assumed, where replacing all congruent temporaries by a single representative preserves the program semantics. LSSA is also referred to as *the really-crude approach* to SSA [2, 5] and *maximal SSA* [15]. LSSA's single definitions and local temporaries are crucial for modeling global register allocation: these properties preserve live ranges as simple intervals, which is required for modeling register assignment and packing geometrically.

After LSSA construction, each basic block in the input function is copy-extended and adjusted for rematerialization as in Section 4. Copy extension induces alternative temporaries that can be used by each operation, making it necessary to lift the LSSA congruence from temporaries to operands as illustrated in Figure 9. Assume that *pred* and *succ* are adjacent basic blocks. After copy extension, each congruence $t_{pred} \equiv t_{succ}$ is lifted to the congruence $p_{pred} \equiv p_{succ}$ where p_{pred} is the exit operand using t_{pred} (or any of its alternatives $t_{pred.1}, \ldots, t_{pred.n}$) and p_{succ} is the entry operand defining t_{succ}.

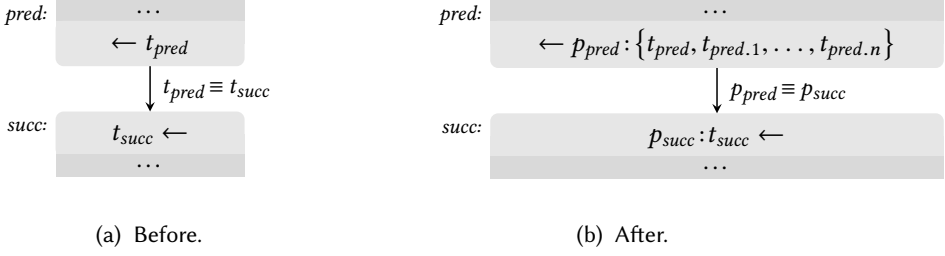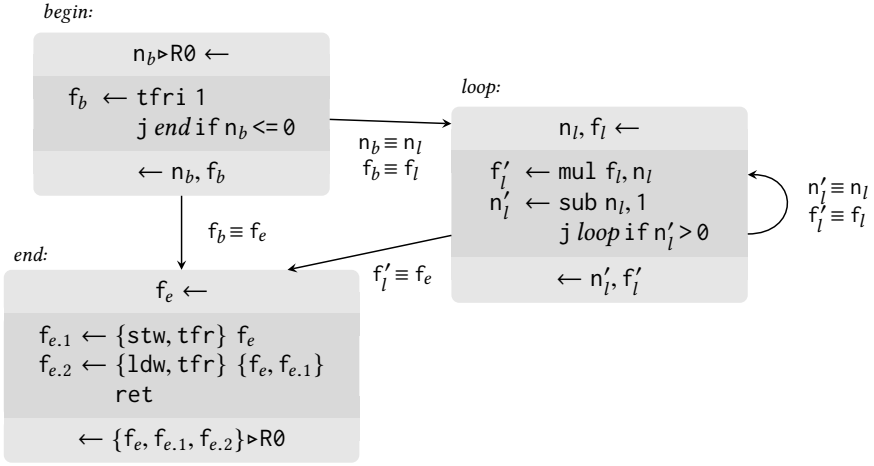(a) Before.   (b) After.

Fig. 9. Lifting of the LSSA congruence from temporaries to operands.

Fig. 10. CFG of the factorial function from Example 2 in LSSA where f_e is copy-extended.

Example 7. Figure 10 shows the input representation of the factorial function from Example 2 for global register allocation. The function is in LSSA: each original temporary t in Example 2 is decomposed into a temporary t_b for each block b in which the temporary is live. For example, the temporary n is decomposed into temporaries n_b for the *begin* basic block and n_l for the *loop* basic block (for conciseness, only the first letter of a block name is used as index). To enforce single definitions, the redefinition of n_l is further renamed as n_l' similarly to Example 3. The resulting temporaries are local and defined only once. For example, n_l is defined once by the entry operation of *loop* and only used locally by mul and sub.

The input representation to global register allocation assumes that all temporaries are copy-extended, but the example limits copy extension to f_e for simplicity, which yields the alternative temporaries $f_{e.1}$ and $f_{e.2}$. Also for simplicity, the LSSA congruence (displayed on the CFG arcs) is defined on temporaries and not on operands as shown in Figure 9.

5.2 Model

The program representation for global register allocation preserves the structure of the local model of each basic block described in Section 4 since operations, operands, and temporaries are local and basic blocks are only related by operand congruences. Therefore, a global model of register allocation is simply composed of the variables and constraints of each local model and linked with

(a) Initial program.   (b) Base case.   (c) Global spilling.
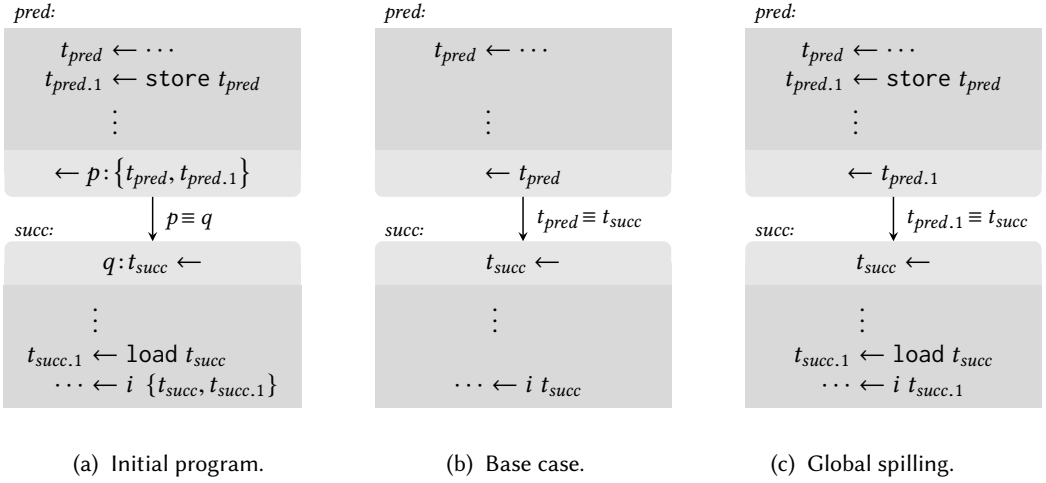
Fig. 11. Interplay of live temporary, register class, and congruence constraints.

constraints that assign congruent operands to the same register:

$$\mathbf{reg}[\mathbf{temp}(p)] = \mathbf{reg}[\mathbf{temp}(q)] \quad \forall p, q \in P : p \equiv q \,. \tag{C6}$$

The congruence constraints extend the scope of all register allocation transformations to whole functions by propagating the impact of local decisions across basic blocks. Figure 11 illustrates the interplay of live temporary (C5), register class (C3.2), and congruence (C6) constraints across two basic blocks *pred* and *succ*.

Figure 11a displays a program fragment where the original temporary t is live across the boundaries of *pred* and *succ*. Temporary t is decomposed into t_{pred} and t_{succ} and copy-extended with a store-move copy operation (store) at *pred* and a load-move copy operation (load) at *succ* (the remaining copy operations and register-to-register move instructions are omitted for simplicity).

Figure 11b shows the base case where t_{pred} is not spilled (hence the store-move is deactivated). Since $t_{pred.1}$ is dead, the exit operation of *pred* uses t_{pred} and the congruence constraints propagate the processor register of t_{pred} to t_{succ}. Now the processor register of t_{succ} is incompatible with the register class of load, which deactivates the load-move and forces i to use t_{succ}.

Figure 11c shows the global spill of t where the store-move is active and the exit operation of *pred* uses $t_{pred.1}$ instead of t_{pred}. In this case the congruence constraints propagate the memory register of $t_{pred.1}$ to t_{succ}, which forces i to use the temporary $t_{succ.1}$ copied from t_{succ} by the load-move.

In general, entry and exit operations define and use multiple temporaries. All temporaries used or defined by the same boundary operation interfere and are thus necessarily assigned to different registers according to constraint C1.1. The use of LSSA in its conventional form [92] guarantees the absence of conflicts between this and the congruence constraints (C6).

The same rectangle assignment and packing constraint defined for a single basic block in Section 4.2 is used for each basic block:

$$\text{no-overlap}\,(\{\langle \mathbf{reg}(t), \mathbf{reg}(t) + width(t), \mathbf{start}(t), \mathbf{end}(t)\rangle : t \in T_b \ \wedge \mathbf{live}(t)\}) \quad \forall b \in B\,. \tag{C1.1}$$

Example 8. Figure 12 illustrates global register allocation for the input function from Example 7. It corresponds to the better solution in the introductory Example 1: the temporaries derived from n are assigned to R0 and the temporaries derived from f are assigned to R1 with the exception of $f_{e.1}$.
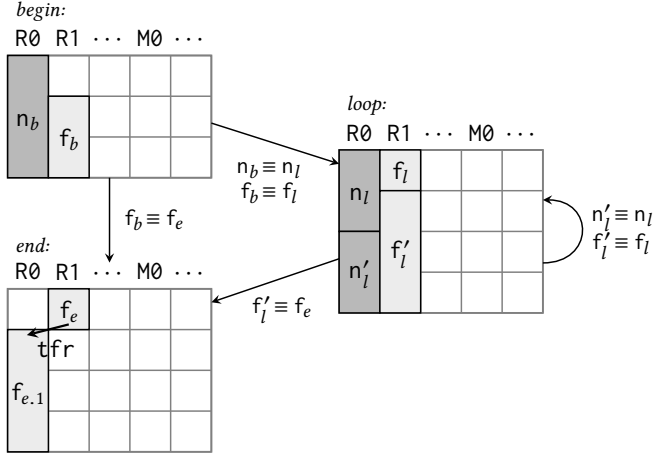
Fig. 12. Global register allocation for the factorial function from Example 7.

$f_{e.1}$ is copied to R0 in the *end* basic block by activating its defining store-move and implementing it with the `tfr` instruction. The load-move of f_e is inactive, which renders $f_{e.2}$ dead (C5).

The congruence constraints (C6) guarantee the compatibility of the local register assignments across boundaries. For example, the live-out temporaries of *begin* ($\langle n_b, f_b \rangle$) and the live-in temporaries of *loop* ($\langle n_l, f_l \rangle$) are pairwise congruent and thus assigned to the same registers ($\langle R0, R1 \rangle$).

Discussion. The introduced model is the first to exploit the properties of LSSA for register allocation. LSSA and similar representations have been used to establish theoretical connections between programming paradigms [2], as an intermediate form in SSA construction [5], and as a basis for intermediate-level program analysis and optimization [40].
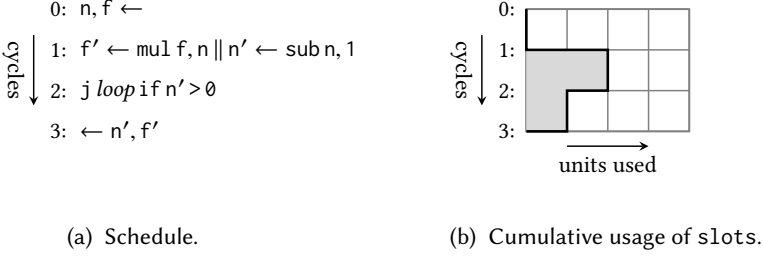
The model has a limitation in that it does not support multi-allocation across basic blocks. This is due to the fact that only one temporary among possibly many copied temporaries can be used by exit operands as seen in Figure 9. This limitation is shared with most combinatorial register allocation approaches (see Table 1), and its impact on scalability and code quality is unclear. Among integrated approaches (Table 2), only Wilson *et al.*'s model [101] is global. Wilson *et al.* propose a similar concept to congruence constraints but their model does not support multi-allocation.

Multi-allocation in a combinatorial setup is discussed in detail by Colombet *et al.* [31]. Among the three scenarios discussed by Colombet *et al.*, our model supports the optimizations illustrated in Figures 1 and 2 from Colombet *et al.*'s paper [31] but not in Figure 3.

6   INSTRUCTION SCHEDULING

This section describes the model for local instruction scheduling. It is mostly based on previous work on constraint-based scheduling by Malik *et al.* [67] and serves as an intermediate step towards the integrated register allocation and instruction scheduling model. It captures pre-register allocation scheduling and assumes the same basic block representation as in Section 4, with the only difference that operations are not totally ordered. Post-register allocation scheduling can be easily captured by introducing additional dependencies caused by register assignment.

Variables. Instruction scheduling assigns issue cycles to operations in a basic block such that dependencies between operations and processor resource constraints are satisfied. In the model, a

(a) Schedule.

(b) Cumulative usage of slots.

Fig. 13. Instruction schedule for the *loop* basic block from Example 3.

variable **issue**(o) defines the cycle in which o is issued relative to the beginning of the basic block:

$$\textbf{issue}(o) \in \mathbb{N}_0 \quad \forall o \in O. \tag{V6}$$

Dependency constraints. Valid schedules must preserve the input order among dependent operations. An operation u depends on another operation d if u uses a temporary t defined by d. If operation d is issued at cycle **issue**(d), its result becomes available at cycle **issue**(d) + $lat(ins(d))$, where $ins(o)$ is the instruction that implements operation o and $lat(i)$ is the latency of instruction i. To satisfy the dependency, u must be issued after the result of d is available, that is: **issue**(u) $\geq$ **issue**(d) + $lat(ins(d))$. The model includes one such inequality for each use of a temporary t, where u uses t through operand q and d defines t through operand p:

$$\textbf{issue}(operation(q)) \geq \textbf{issue}(operation(p)) + lat(ins(operation(p)))$$
$$\forall t \in T, \forall p \in \{definer(t)\}, \forall q \in users(t) : temp(q) = t. \tag{C7}$$

The dependency constraints deviate slightly from early constraint models such as that of Malik *et al.* [67] in that they make the underlying uses and definitions of temporaries explicit. This is essential for the integration with register allocation as explained in Section 7. The integration also requires that the virtual entry instruction is given a latency of one. This requirement ensures that live-in temporaries in a basic block have non-empty live ranges and are thus assigned to different registers according to constraint C1.1.

Processor resource constraints. The capacity of processor resources such as functional units and buses cannot be exceeded. As discussed in Section 2.3, this paper assumes a set of processor resources S where each resource $s \in S$ has a capacity $cap(s)$ and each instruction i consumes $con(i, s)$ units of each resource s during $dur(i, s)$ cycles. The model includes a *cumulative* constraint for each resource s to ensure that the capacity of s is never exceeded by the scheduled operations:

$$\text{cumulative}\left(\{\langle \textbf{issue}(o), dur(ins(o), s), con(ins(o), s)\rangle : o \in O\}, cap(s)\right) \quad \forall s \in S. \tag{C8}$$

This paper assumes that resource consumption always starts at the issue of the consumer. The model can be easily extended to capture more complex scenarios (for example, modeling each stage in a processor pipeline) by adding an offset $off(i, s)$ to the issue cycle of the consumer (**issue**(o)) in the resource constraint (C8) [19].

Example 9. Figure 13a shows a schedule of the *loop* basic block from Example 3 where all instructions are assumed to have a latency of one cycle. The vertical dimension ranges over cycles as opposed to program points in register allocation.

Hexagon is a VLIW processor that allows up to four instructions to be issued in parallel. In the example, only mul and sub can be issued in parallel ($\parallel$) as they do not depend on each other. The multiple-issue capacity of Hexagon is modeled as a resource slots with $cap(\text{slots}) =$

4, $con(i, \texttt{slots}) = 1$, and $dur(i, \texttt{slots}) = 1$ for each instruction $i \in \{\texttt{mul}, \texttt{sub}, \texttt{j if}\}$. With this resource, constraint C8 is instantiated as cumulative ($\{\langle\textbf{issue}(\texttt{mul}), 1, 1\rangle, \langle\textbf{issue}(\texttt{sub}), 1, 1\rangle, \langle\textbf{issue}(\texttt{j if}), 1, 1\rangle\}, 4$). Figure 13b shows the cumulative usage of the slots resource over time. The figure shows that parallelism in the example is not limited by the slots resource constraint (C8) but by the dependency constraint $\textbf{issue}(\texttt{j if}) \geq \textbf{issue}(\texttt{sub}) + 1$ over temporary n′ (C7).

Discussion. Constraint programming is a popular technique for resource-constrained scheduling problems [6], as it provides efficient and dedicated propagation algorithms for scheduling constraints. As shown here, it enables a simple model using a single variable per operation and a single constraint for each dependency and resource. For comparison, integer programming typically requires a decomposition into n variables with $\{0, 1\}$ domain for each operation and n linear constraints for each resource, where n is an upper bound on the number of issue cycles of the basic block. This decomposition limits the scalability of approaches based on integer programming [24].

7 GLOBAL REGISTER ALLOCATION AND INSTRUCTION SCHEDULING

This section combines the models for global register allocation and instruction scheduling into an integrated model that precisely captures their interdependencies. The integrated model is a significant contribution to combinatorial code generation as, for the first time, it captures the same program transformations as state-of-the-art heuristic approaches (see Table 2).

The model assumes the program representation described in Section 5.1, except that operations within basic blocks are not totally ordered. The model uses the variables and constraints from the global register allocation model and the instruction scheduling model generalized to handle copy-extended programs. This generalization allows copy operations (introduced to support spilling, live range splitting, and rematerialization) to be scheduled in the same manner as the original program operations, by means of issue cycle variables. Compared to the register allocation model without instruction scheduling, the ability to schedule and thus rearrange copy operations increases the freedom of their supported program transformations.

Live ranges and issue cycles. In the integrated model, live ranges link register allocation and instruction scheduling since they relate registers assigned to temporaries with the issue cycles of their definers and users. The live start of a temporary t is linked to the issue of the operation d defining t, and the live end is linked to the issue of the last operation u_n using t as shown in Figure 14. Variables $\textbf{start}(t)$ and $\textbf{end}(t)$ are introduced for each temporary t giving its live start and end cycles:

Fig. 14. Live range of t.

$$\textbf{start}(t), \textbf{end}(t) \in \mathbb{N}_0 \quad \forall t \in T. \tag{V7}$$

The live start of a (live) temporary t corresponds to the issue of its defining operation:

$$\textbf{start}(t) = \textbf{issue}(operation(definer(t))) \quad \forall t \in T : \textbf{live}(t), \tag{C9}$$

while the live end of t corresponds to the issue of the last user operation:

$$\textbf{end}(t) = \max_{p \in users(t)\,:\,\textbf{temp}(p)\,=\,t} \textbf{issue}(operation(p)) \quad \forall t \in T : \textbf{live}(t). \tag{C10}$$

Generalized instruction scheduling. The generalized model captures alternative instructions, optional (active or inactive) operations, and alternative temporaries. Alternative instructions are handled by using the variable $\textbf{ins}(o)$ instead of the parameter $ins(o)$ for each operation o. The generalized dependency constraints handle inactive operations and alternative temporaries by making each dependency involving an operation that potentially uses temporary t through

Fig. 15. Integrated solution for the factorial function from Example 7.

operand q conditional on whether the operation is active and actually uses t:

$$\textbf{issue}(operation(q)) \geq \textbf{issue}(operation(p)) + lat\,[\textbf{ins}(operation(p))]$$

$$\forall t \in T, \forall p \in \{definer(t)\}, \forall q \in users(t) : \textbf{active}(operation(q)) \wedge \textbf{temp}(q) = t \,. \tag{C7.1}$$

The generalized processor resource constraints handle optional operations by making their resource usage conditional on whether they are active. Additionally, they are extended to global scope by including a *cumulative* constraint for each resource s and basic block b:
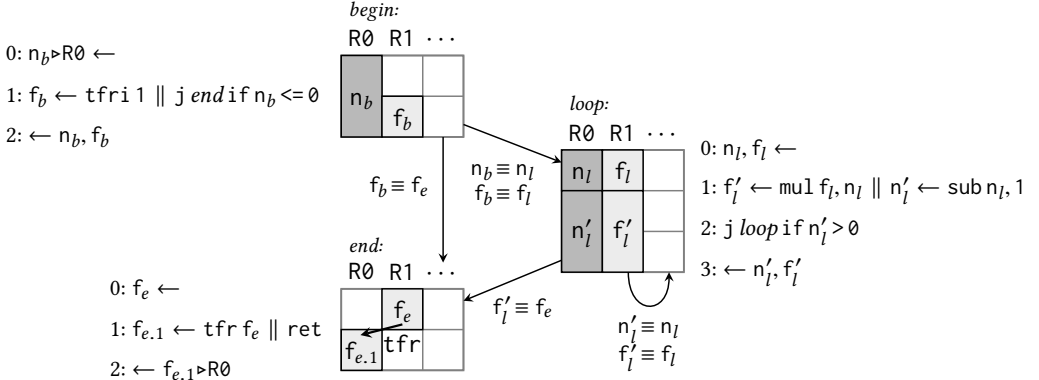
$$\text{cumulative}\,(\{\langle\textbf{issue}(o), dur\,[\textbf{ins}(o), s]\,, con\,[\textbf{ins}(o), s]\rangle : o \in O_b \wedge \textbf{active}(o)\}, cap(s))$$

$$\forall b \in B\,, \forall s \in S. \tag{C8.1}$$

Example 10. Figure 15 shows a solution to the integrated register allocation and instruction scheduling problem for the function in Example 7. The scheduled operations are shown besides the register assignment of each basic block. The memory registers $M0, M1, \ldots$ are omitted for conciseness. The solution corresponds to the faster assembly code given in Example 1. The final assembly code shown in Example 1 can be generated by simply removing the entry and exit virtual operations and replacing each temporary with its assigned register.

Discussion. While the scope of register allocation in the integrated model is global, instruction scheduling is local. This limitation is shared with the only available integrated approach that is global (Wilson *et al.* [101], see Table 2). Extending instruction scheduling beyond basic blocks is known to improve code quality, particularly for VLIW processors [39]. A number of combinatorial approaches have explored such extensions successfully [24]. The model presented in this section might be readily expanded to superblocks (consecutive basic blocks with multiple exit points but a single entry point) using a corresponding number of exit operations similarly to Malik *et al.* [66].

8 OBJECTIVE FUNCTION

This section completes the integrated model with a generic objective function. The entire model, including the objective function, is presented in Appendix A.

The objective function is orthogonal to the variables and constraints in the model and can be chosen to optimize for different goals. In this paper, the objective is to minimize the weighted sum

of the cost of each basic block:

$$\sum_{b \in B} weight(b) \times \textbf{cost}(b) \tag{5}$$

where $weight(b)$ and $\textbf{cost}(b)$ can be defined to optimize for different goals.

Optimizing speed. To optimize for speed, the weight of a basic block b is set as its execution frequency $freq(b)$ and its cost is defined as the issue cycle of the exit operation of b (indicated by $exit(b)$), which by construction is the last scheduled operation in b:

$$weight(b) = freq(b); \qquad \textbf{cost}(b) = \textbf{issue}(exit(b)) - 1 \quad \forall b \in B. \tag{6}$$

Note that subtracting one to $\textbf{issue}(exit(b))$ accounts for the cycle occupied by the virtual entry operation, as discussed in Section 6.

Optimizing code size. To minimize code size, it is sufficient to choose one as weight for all blocks, as they contribute proportionally to the total code size. The cost is defined as the sum of the size of the instruction i (given by $size(i)$) implementing each active operation:

$$weight(b) = 1; \qquad \textbf{cost}(b) = \sum_{o \in O_b \,:\, \textbf{active}(o)} size[\textbf{ins}(o)] \quad \forall b \in B. \tag{7}$$

Other optimization criteria such as spill code or energy consumption minimization can be expressed by defining $weight(b)$ and $\textbf{cost}(b)$ accordingly. Constraint programming also supports optimizing for multiple criteria (for example, first for speed and then for code size) by using a tuple as the objective function.

Example 11. Assuming the estimated execution frequencies $freq(begin) = 1$, $freq(loop) = 10$, and $freq(end) = 1$ and a uniform size of 32 bits for all real instructions, the value of the speed objective function (equation (6)) for the solution given in Example 10 is:

$$1 \times (2 - 1) + 10 \times (3 - 1) + 1 \times (2 - 1) = 1 + 20 + 1 = 22 \text{ execution cycles}, \tag{8}$$

while the value of the code size objective function (equation (7)) is:

$$1 \times 64 + 1 \times 96 + 1 \times 64 = 64 + 96 + 64 = 224 \text{ bits}. \tag{9}$$

Discussion. The cost model that underlies the objective function for speed optimization (equation (6)) assumes a processor with constant instruction latencies. This assumption, common in combinatorial approaches, may underestimate the contribution of variable latencies to the actual execution time. Generally, the less predictable the targeted processor is, the lower the accuracy of the speed cost model. In extreme cases (out-of-order, superscalar, speculative processors with long pipelines and deep memory hierarchies), other optimization criteria such as spill code minimization might be chosen as a better approximation to speed optimization.

9 ADDITIONAL PROGRAM TRANSFORMATIONS

A key advantage of combinatorial approaches is the ease with which additional program transformations and processor-specific features can be captured in a compositional manner. This section contributes model extensions to handle program transformations that depend on register allocation and instruction scheduling but are usually approached in isolation by today's state-of-the-art heuristic compilers.

Stack frame elimination. One of the main responsibilities of code generation is to manage the call stack by placing special operations at entry, return, and function call points. *Stack frame elimination* is a common optimization that avoids generating a stack frame for spill-free functions that meet some additional conditions (for example not containing calls or other stack-allocated temporaries). Heuristic compilers typically apply this optimization *opportunistically* after deciding whether to spill in register allocation. In contrast, our approach captures the optimization in integration with register allocation and instruction scheduling, hence taking into account the overhead in generating a frame by spilling in functions where the frame could be otherwise avoided.

The model is easily extended to capture stack frame elimination by introducing a single variable **frame** that indicates whether the function under compilation requires a frame:

$$\textbf{frame} \in \mathbb{B}. \tag{V8}$$

This variable is true if there exists an active operation o implemented by an instruction i requiring a frame (indicated by *requires-frame*(o, i)):

$$\exists o \in O : \textbf{active}(o) \wedge \textit{requires-frame}[o, \textbf{ins}(o)] \implies \textbf{frame}. \tag{C11}$$

Typical examples of operations requiring a frame are calls and copy operations implemented by spill instructions. If the function requires a frame, each optional operation o implemented by instruction i managing the stack (indicated by *manages-frame*(o, i)) must be active:

$$\textbf{frame} \implies \textbf{active}(o) \quad \forall o \in O : \textit{manages-frame}[o, \textbf{ins}(o)]. \tag{C12}$$

Operations that manage the stack typically include pushes, pops, and adjustments of the pointers that delimit the stack frame.

Scheduling with latencies across basic blocks. Local instruction scheduling as described in Section 6 conservatively assumes that a temporary defined within a certain basic block might be used at the beginning of a successor basic block. This assumption can force long-latency instructions such as integer divisions to be scheduled unnecessarily early, which limits the solution space. While this paper is confined to local instruction scheduling, latencies across basic blocks (including loops) are captured by a simple model extension.

Assume a temporary t live across basic blocks before LSSA construction. The key idea is to distribute the latency of t between the use and the definition of the corresponding LSSA temporaries at the basic block boundaries. The operands of virtual boundary operations that are not placed at the function boundaries are called *boundary operands*. A boundary operand p can be classified as either entry or exit (indicated by *entry-operand*(p) or *exit-operand*(p)). A variable **slack**(p) for each boundary operand p gives the amount of latency *slack* assigned to p. The slack can be either zero (no latency across boundaries) or negative for exit operands and positive for entry operands. The latter case corresponds to a temporary that is defined *late* in the predecessor basic block:

$$\textbf{slack}(p) \in \begin{cases} \{0\} \cup \mathbb{Z}^-, & \textit{exit-operand}(p) \\ \{0\} \cup \mathbb{Z}^+, & \textit{entry-operand}(p) \quad \forall p \in P, \\ \{0\}, & \text{otherwise} \end{cases} \tag{V9}$$

To avoid that pre-LSSA temporaries defined late in a basic block are used too early in successor basic blocks, the slack of the boundary operands relating the derived LSSA temporaries across basic blocks boundaries must be balanced:

$$\textbf{slack}(p) + \textbf{slack}(q) = 0 \quad \forall p, q \in P : p \equiv q. \tag{C13}$$
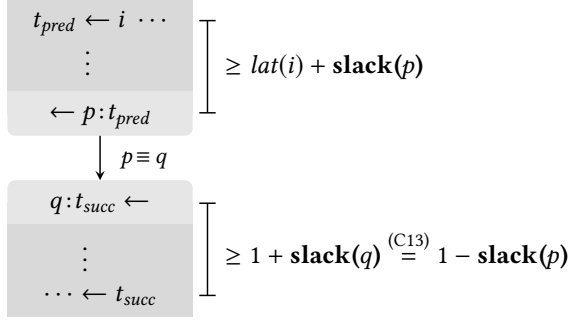
Fig. 16. Distribution of latency slack among the boundaries of two adjacent basic blocks.

Finally, the latency added or subtracted by the slack is reflected in the dependency constraints:

$$\mathbf{issue}(operation(q)) \geq \mathbf{issue}(operation(p)) + lat[\mathbf{ins}(operation(p))] + \mathbf{slack}(p) + \mathbf{slack}(q)$$

$$\forall t \in T, \forall p \in \{definer(t)\}, \forall q \in users(t) : \mathbf{active}(operation(q)) \wedge \mathbf{temp}(q) = t.$$

(C7.2)

Figure 16 illustrates how the slack is distributed among the boundary operands relating the congruent temporaries t_{pred} and t_{succ}. Because of the slack balancing constraints (C13), the slack of operand p is the opposite of the slack of q. Note that the latency of the entry operation in the successor basic block is one as discussed in Section 6.

Scheduling with operand forwarding. Operand forwarding is a processor optimization that makes a value available before the end of the execution of its definer instruction. Some processors such as Hexagon exploit this optimization by providing operand-forwarding instructions that access values from registers in the same cycle as they are defined. These instructions reduce the latency of their dependencies but may impose additional constraints on instruction scheduling, hence it is desirable to capture their effect in the model.

An operation implemented by an instruction i forwarding its operand p (which is indicated by *forwarded*(i, p)) is scheduled in parallel with the definer of the temporary used by p:

$$\mathbf{issue}(o) = \mathbf{issue}(operation(\mathbf{temp}(p)]))$$

$$\forall p \in P, \forall o \in \{operation(p)\} : \mathbf{active}(o) \wedge forwarded[\mathbf{ins}(o), p].$$

(C14)

If an operand is forwarded, its corresponding dependency constraint that includes the latency of the definer instruction has no effect:

$$\mathbf{issue}(operation(q)) \geq \mathbf{issue}(operation(p)) + lat[\mathbf{ins}(operation(p))]$$

$$\forall t \in T, \forall p \in \{definer(t)\}, \forall q \in users(t) :$$

$$\mathbf{active}(operation(q)) \wedge \mathbf{temp}(q) = t \wedge \neg forwarded[\mathbf{ins}(operation(q)), q].$$

(C7.3)

For clarity, this refinement of the dependency constraints is presented independently of that required to capture latencies across basic blocks (C7.2), but both are compatible.

Selection of two- and three-address instructions. Two-address instructions are instructions of the form Rd ← i Rd, Rs where the register Rd gets overwritten. These instructions can often be encoded more compactly but impose additional constraints on register allocation. In particular, the definition and use operands p, q that correspond to the register overwrite of a two-address
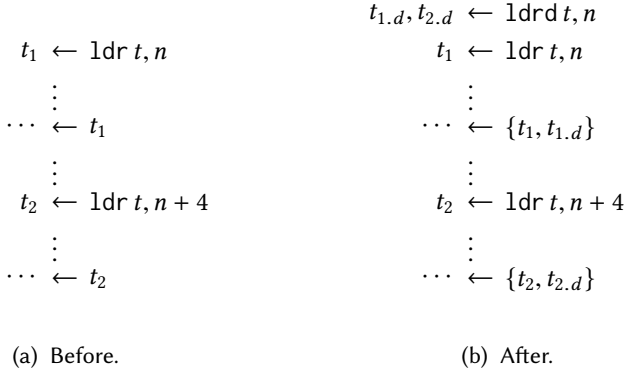
$$t_{1.d}, t_{2.d} \leftarrow \texttt{ldrd}\, t, n$$

$$t_1 \leftarrow \texttt{ldr}\, t, n \qquad\qquad\qquad t_1 \leftarrow \texttt{ldr}\, t, n$$

$$\vdots \qquad\qquad\qquad\qquad\qquad \vdots$$

$$\cdots \leftarrow t_1 \qquad\qquad\qquad \cdots \leftarrow \{t_1, t_{1.d}\}$$

$$\vdots \qquad\qquad\qquad\qquad\qquad \vdots$$

$$t_2 \leftarrow \texttt{ldr}\, t, n + 4 \qquad\qquad t_2 \leftarrow \texttt{ldr}\, t, n + 4$$

$$\vdots \qquad\qquad\qquad\qquad\qquad \vdots$$

$$\cdots \leftarrow t_2 \qquad\qquad\qquad \cdots \leftarrow \{t_2, t_{2.d}\}$$

(a) Before.   (b) After.

Fig. 17. Extension for double-load instruction selection.

instruction i (indicated by *two-address*(i, p, q)) are assigned to the same register:

$$\mathbf{reg}[\mathbf{temp}(p)] = \mathbf{reg}[\mathbf{temp}(q)]$$

$$\forall p, q \in P, \forall o \in \{operation(p)\} : \mathbf{active}(o) \wedge \textit{two-address}[\mathbf{ins}(o), p, q]. \tag{C15}$$

Some instruction sets like ARM's Thumb-2 allow two- and three-address instructions to be freely intermixed. The selection of two- and three-address instructions is captured by constraint C15 together with alternative instructions as introduced in Section 4.3.

Selection of double-load and double-store instructions. ARM v5TE and later versions provide load (`ldrd`) and store (`strd`) instructions that access two 32-bits values at consecutive addresses in memory. Combining pairs of 32-bits load (`ldr`) or store instructions (`str`) into their double counterparts can improve memory bandwidth but might increase register pressure by making the two 32-bits values interfere. Single/double-load instruction selection can be integrated into the model using alternative temporaries (the process is analogous for double-store instructions).

The input program is extended with an optional `ldrd` operation for each pair of `ldr` operations accessing consecutive addresses ($t + n, t + n + 4$), and the temporaries loaded by `ldrd` are presented as alternatives to the temporaries loaded by each individual `ldr`. Figure 17 illustrates the extension.

Nandivada and Palsberg [73] propose exploiting double-store and double-load instructions for spilling. Incorporating this feature into our model is in principle possible since memory registers capture the assignment of individual stack frame locations.

10   SOLVING IN UNISON

Our integrated, combinatorial approach to register allocation and instruction scheduling is implemented in the *Unison* software tool [23]. The core feature of Unison is its use of constraint programming (CP) [83] to exploit the structure of these compiler problems. This section outlines the main methods used to scale to medium-sized problems, as well as Unison's implementation. The practical impact of these methods is studied in Section 11.3.

10.1   External Solver Portfolio

Constraint solvers guarantee to find the optimal solution to a problem if there is one. However, they typically exhibit a large variability in solving time, even for problems of similar size [44]. Furthermore, different solvers tend to perform best for problems of different sizes and structure, particularly when the solving methods differ. This variability can be exploited to reduce solving

time by running a portfolio of multiple solvers in parallel on the same problem [43]. Unison employs two different portfolios, an *internal* and an *external* portfolio. The internal portfolio is discussed in Section 10.2. The external portfolio runs two solvers in parallel each using a single operating system process: a *decomposition-based* solver that exploits the problem structure for scalability and an off-the-shelf hybrid CP-Boolean satisfiability solver. As Section 11.3 discusses, the two solvers complement each other and thus yield a portfolio that is superior to each of them in isolation.

Decomposition-based solver. The decomposition-based solver is based on a novel use of the LSSA form (see Section 5.1) for increased scalability and anytime behavior. The key insight is that, in the combinatorial model of a LSSA function, the only connection between basic blocks is established by the congruence constraints (C6) on boundary operands. If registers are assigned to all boundary operands, the rest of the register allocation and instruction scheduling problem can be decomposed and solved independently for each basic block.

The decomposition-based solver exploits this structure to solve the combinatorial problems iteratively at two levels: first, a *master problem* is solved where **reg(temp(p))** is assigned for each boundary operand p such that no constraint violation is detected by propagation. If the model is extended as in Section 9, **frame** and **slack(p)** for each boundary operand p must also be assigned as part of the master problem. Then, a *subproblem* is solved for each basic block b by assigning its remaining variables such that all but the congruence constraints (C6) are satisfied and **cost(b)** is minimized. Finally, the solution m to the master problem and the solutions to all subproblems are combined into a full solution s, the value of s is evaluated according to equation (5), and a new iteration is run. In the new iteration, the objective function is constrained to be less than that of s and ¬m is added to the set of constraints to avoid finding the same solution to the master problem again. When the solver proves optimality or times out, the last full solution is returned.

10.2   Solving Improvements

The model introduced through Sections 4-9 can be directly implemented and solved with CP. However, as is common in combinatorial optimization a wide array of additional modeling and solving improvements are required in practice to scale beyond small problems. This section outlines the most successful methods used in Unison.

Implied constraints. An effective improving method in CP is to add *implied constraints*, which are constraints that are logically redundant but yield additional propagation hence reducing the amount of search required [88]. Implied constraints in Unison include:

- Different temporaries used by an operation must be assigned to different registers.
- The live range of the temporary used by the first operand of an operation o implemented by a two-address instruction (see Section 9) ends at the issue cycle of o.
- If definition operands p, q are pre-assigned to the same register and it is known statically that **issue($operation(p)$) < issue($operation(q)$)**, then **end[temp(p)] ≤ start[temp(q)]** holds.
- *Cumulative constraints* derived from the interference constraints (C1.1) where the live range of each temporary t yields a task that consumes *width(t)* units of a register resource [87].
- *Distance constraints* of the form **issue(o_2) ≥ issue(o_1) + $d(o_1, o_2)$** derived from a static *region analysis* [67, 100], where operation o_1 is statically known to precede operation o_2 and their derived issue cycle distance $d(o_1, o_2)$ is greater than the distance that would be otherwise inferred by propagation of the dependency constraints.
- Constraints enumerating all allowed combinations of temporaries used or defined by sets of copy-related operands. For each such set, the allowed combinations of used and defined

temporaries are derived from a relaxation of the combinatorial model that only includes the constraints over **active**(o), **live**(t), and **temp**(p).

- *Cost lower bound constraints* of the form **cost**(b) ≥ c where c is the optimal cost to a relaxation of the subproblem for basic block b (see Section 10.1) where the registers of the boundary operands are unassigned.

Symmetry breaking constraints. Combinatorial models often admit multiple solutions that are *symmetric* in that they are all considered equivalent [88]. Example sources of symmetries in our model are *interchangeable registers* (where the registers assigned to two temporaries can be swapped) and *interchangeable copy operations* (where the copy operations that support a spill or a live range split can be swapped). A common improving method is to add *symmetry breaking constraints* that reduce the search effort by avoiding searching for symmetric solutions. Symmetries caused by both interchangeable registers and copy operations are broken in Unison with global *value precedence constraints* [60], which enforce a precedence among values in a sequence of variables.

Dominance breaking constraints. Combinatorial models often admit solutions that are *dominated* in that they can be mapped to solutions that are always of at most the same cost. Similarly to symmetries, dominated solutions can be discarded by *dominance breaking constraints* to reduce the search effort [88]. An example of dominance in our model is in the registers of the source t_s and destination t_d temporaries of an active copy operation o: for o to be useful, constraints are added to enforce the assignment of t_s and t_d to different registers. Another example involves the variables related to inactive operations (issue cycles, instructions, and temporaries) and dead temporaries (registers and live range cycles): such variables are assigned to arbitrary values as they do not affect the constraints in which they occur.

Probing. Probing is a common method in combinatorial optimization that tests variable assignments and discards the values for which propagation proves the absence of solutions [84]. The method is performed only before search starts, as it has a high computational cost and its benefit is highest at this phase. Probing is applied both to the instructions of **ins**(o) for each operation o and to the values of the objective function (equation (5)) and **cost**(b) for each basic block b.

Internal portfolio. The decomposition-based solver discussed above solves each subproblem with a portfolio of *search strategies*. A search strategy defines how a problem is decomposed into alternative subproblems and the order in which search explores the subproblems. The strategies included in the internal portfolio complement each other (hence achieving better robustness) and differ in two main aspects. First, they differ in the order in which they select variables to try values on. For example, some strategies try to assign instructions and temporaries for one operation at a time while others try to assign instructions for all operations first. Second, the strategies differ in the order in which they select values to be tried for a variable. For example, different strategies try to schedule operations either as early as possible or as close as possible to their users. Each search strategy runs as a portfolio asset, but in contrast to the external portfolio the assets communicate by adding constraints to their search based on the cost of solutions found by the other assets.

10.3 Implementation

Unison is implemented as an open-source tool [23] that can be used as a complement to the LLVM compiler [59]: the LLVM interface, the transformations to the input program (such as LSSA construction and copy extension), and the processor and calling convention descriptions are implemented in Haskell; the decomposition-based solver and the solving improvements are implemented in C++ using the constraint programming system Gecode [42]; and the off-the-shelf solver Chuffed [29] is interfaced through the MiniZinc modeling language [76]. The tool includes in

total 16 569 and 19 456 lines of hand-written Haskell and C++ code. Each of the supported processors is described in around one thousand lines of hand-written Haskell code and complemented by tens of thousands of lines of Haskell code automatically generated from LLVM's processor descriptions. Further detail about Unison's architecture and interface can be found in its manual [19].

11   EXPERIMENTAL EVALUATION

This section presents experimental results for Unison, with a focus on scalability and code quality. Section 11.1 defines the setup of the experimental evaluation, Sections 11.2 and 11.3 analyze the estimated code quality and scalability of Unison for different goals and processors, and Section 11.4 presents a study of the actual speedup achieved for Hexagon V4 on MediaBench applications.

11.1   Setup

Evaluation processors and benchmarks. The experimental evaluation includes three processors with different architectures and application domains: Hexagon V4 [30] (a four-way VLIW digital signal processor), ARM1156T2F-S [4] (a single-issue general-purpose processor), and a generic MIPS32 processor [70] (a single-issue general-purpose processor used predominantly in embedded applications). The specifics of the MIPS32 processor are defined as in the LLVM 3.8 compiler [59]. The rest of the section refers to these processor as *Hexagon*, *ARM*, and *MIPS*. The three processors are in-order, although they include unpredictable features such as cache memories and branch predictors. Evaluating Unison for out-of-order processors such as x86 [52] is part of future work.

The evaluation uses functions from the C applications in MediaBench [61] and SPEC CPU2006 [90]. These benchmark suites match the application domain of the selected processors and are widely employed in embedded and general-purpose compiler research. The perlbench application in SPEC CPU2006 is excluded due to cross-compilation problems [91].

Baseline. The state-of-the-art heuristic compiler LLVM 3.8 is used both to generate the input low-level IR and as a baseline for comparison. LLVM performs register allocation and instruction scheduling heuristically by priority-based coloring [28] and list scheduling [82], and applies the additional program transformations described in Section 9 opportunistically. The cost of LLVM's solution minus one is used as an upper bound on Unison's objective function (equation (5)), which guarantees that any solution found by Unison improves that of LLVM. The execution frequency of each basic block is estimated using LLVM's standard analysis.

Different flags are used to compile for speed and code size optimization. Table 3 shows the flags used for each goal and LLVM component (clang, opt, and llc are LLVM's front-, middle-, and back-end). The neutral optimization flag O2 is chosen for clang to avoid favoring one optimization goal at the expense of the other. O2 is also chosen for code size optimization with llc since this LLVM component does not provide a more specific Oz flag for this goal. Certain CFG transformations that are orthogonal to the problems solved by Unison are disabled from llc to produce a more accurate comparison (see last row in Table 3). The *clustering* column is discussed in Section 11.2.

Unison configuration. Unison is run on a Linux machine equipped with an Intel Xeon E5-2680 processor with 12 hyperthreaded cores and 64 GB of main memory. Each of the solvers in the external portfolio runs in its own operating system process with a time limit of 15 minutes. The time limit has been chosen to make it feasible to run a large number of benchmarks. Time has been chosen as a limit as other measures (such as number of search nodes) vary widely across the benchmark set. This limit does not only determine the solving time but also the quality of the generated code: as the solvers exhibit anytime behavior, a larger time limit translates into better code. The potential dispersion in code quality measurements due to timeouts is controlled by using

Table 3. Flags for each goal and LLVM component.

	speed optimization	code size optimization	clustering
`clang`	O2	O2	O2
`opt`	O3	Oz	O2
`llc`	O3	O2	O2
	prefixed with `disable-`: `post-ra`, `tail-duplicate`, `branch-fold`, `block-placement`, `phi-elim-edge-splitting`, `if-conversion`, `hexagon-cfgopt` (Hexagon), `arm-optimize-thumb2-in-cp-islands` (ARM), `skip-mips-long-branch` (MIPS)		

the median of 5 repetitions. Across these repetitions, code quality varies for 5% of the functions. For the most variable function, the quality between the worst and the best solutions differs by 21.1%.

To make benchmarking feasible, the internal portfolio in the decomposition-based solver is run sequentially in a round-robin fashion. The same holds true when solving the subproblems. We have verified that running the internal portfolio and solving the subproblems in parallel is only slightly advantageous in that it improves the estimated speedup by a factor of less than 1.1. Instead, the experiments devote the large number of cores available to solving multiple functions in parallel.

The experiments use the revision 33bdc9b of Unison [23], Gecode 6.0.0 [42] for the decomposition-based solver, and Chuffed as distributed with MiniZinc 2.1.2 [69].

11.2 Estimated Code Quality

This section compares the quality of the code generated by Unison to that of LLVM as a state-of-the-art representative of heuristic approaches. The approaches are compared for speed and code size optimization on the three evaluation processors. The quality of Unison and LLVM solutions is estimated according to equation (5). As discussed in Section 8, the speed estimation is subject to inaccuracies due to its dynamic nature, while code size reduction is a static goal which leads to a naturally accurate estimation [58]. Actual speedup measurements are given in Section 11.4, while the accuracy of the speed estimation is studied in detail in Appendix B.

The quality of a Unison solution is presented as its improvement (speedup and code size reduction) over LLVM's corresponding solution. The improvement of a Unison solution where equation (5) has value u over an LLVM solution where equation (5) has value l is computed as:

$$improvement(u, l) = \begin{cases} \frac{l}{u} - 1, & \text{if } l \geq u \\ 1 - \frac{u}{l}, & \text{otherwise.} \end{cases} \tag{10}$$

Hence the magnitude of the improvement is independent of its sign (negative improvements occur in the improvement estimation of isolated approaches and in the actual speedup measurements).

The gap of a Unison solution where the solver provides a lower bound u^* on equation (5) and u and l are defined as above is computed as:

$$gap(u, l, u^*) = improvement(u^*, l) - improvement(u, l) \tag{11}$$

Improvement and gap results are summarized using the geometric mean.

Example 12. According to equation (10), the speedup of Unison over LLVM in Example 1 (assuming the execution frequencies from Example 11) is:

$$improvement(u, l) = improvement(22, 23) = 23 \div 22 - 1 = 4.5\%. \tag{12}$$

Let us assume that the solver times out without proving that the solution in Example 1 is optimal and provides a lower bound of $u^* = 20$. Then the gap is:

$$gap(u, l, u^*) = improvement(20, 23) - improvement(22, 23) = 15\% - 4.5\% = 10.5\%. \qquad (13)$$

Input functions. As input for the estimated code quality and scalability experiments, 100 functions are sampled out of a pool of 10 874 functions from 22 MediaBench and SPEC CPU2006 applications. The purpose of sampling is to keep the experimental evaluation feasible while ensuring that the selected functions are sufficiently diverse to exercise different aspects of Unison. The size of the sampled functions ranges from small (up to 100 input instructions) to medium (between 100 and 1000 input instructions).

The sampling procedure splits all benchmark functions into 100 clusters with a k-means clustering algorithm [65] and selects the most central function from each cluster [79]. Functions are clustered by size (input instructions, input instructions per basic block), key register allocation and instruction scheduling features (register pressure, instruction-level parallelism, and call instruction ratio), and parent application (to improve the spread across the benchmark suites). Register pressure is approximated by the spill ratio of LLVM's register allocator, and instruction-level parallelism is approximated by the number of instructions scheduled per cycle by LLVM's instruction scheduler. The neutral optimization flag 02 is used to extract the clustering features as Table 3 shows. All features are averaged over the evaluation processors. Appendix C details the features and cluster size of each selected function.

Speedup over LLVM. The speedup gained by Unison over LLVM is significant for Hexagon (10% mean speedup), moderate for MIPS (5.4%), and only slight for ARM (1.1%). Figure 18 shows the improvement for each function (black) and the gap for the functions where Unison times out (gray). Each function is identified by a number, the details can be found in Appendix C.

The significant improvement for Hexagon is due to a better exploitation of its VLIW architecture, where Unison generally schedules more instructions in parallel than LLVM's list scheduling algorithm. In doing so, Unison spills on average 28.2% more temporaries, but manages to minimize the overhead by scheduling the spill code in parallel with other instructions. Similarly, the overhead of live range splitting is minimized as its precise cost in terms of instruction scheduling is taken into account. Furthermore, Unison tends to spill less callee-saved registers in loopless functions than LLVM. This type of spilling has a high overhead, because in loopless functions the callee-saved spill code is placed in the most frequent basic blocks. The extreme case is function 72, for which Unison almost doubles the execution speed of LLVM by reducing the number of live range splits by two thirds and the number of callee-saved spills from five to two.

The moderate improvement for MIPS is almost entirely achieved on loopless functions, by spilling short live ranges in low-frequency basic blocks rather than callee-saved registers. For example, Unison speeds up function 79 by 82% by spilling 17 temporaries in low-frequency basic blocks instead of eight callee-saved registers in the high-frequency entry and exit basic blocks. On average, this strategy spills 15.8% more temporaries but reduces the total spill code overhead by 49.1%.

For MIPS on functions with loops and for ARM in general, LLVM's heuristic approach suffices to generate code of near-optimal speed. The occasional speedups achieved for ARM are due to reducing the amount of spilling by 7.5% on average (for example in functions 77, 79 and 100), adjusting the aggressiveness of coalescing to the frequency of the corresponding basic block (for example in functions 77, 79 and 55), and rematerializing more aggressively (for example in function 88).

Code size reduction over LLVM. The code size reduction achieved by Unison over LLVM is moderate for MIPS (3.8% mean code size reduction) and ARM (2.5%), and only slight for Hexagon (1.3%). Although the estimated code size reduction is in general modest, the results directly translate

Fig. 18. Estimated speedup over LLVM (black) and gap (gray) for each function.

into actual improvement unlike the case of speed optimization. Figure 19 shows the improvement for each function (black) and the gap for the functions where Unison times out (gray).

The moderate improvement for MIPS is mostly due to aggressive coalescing that is boosted by the ability to reorder instructions simultaneously. In the extreme case (function 88), Unison eliminates all 10 register-to-register move instructions placed by LLVM. Unison recognizes the high impact of coalescing on code size and spills 5.5% more often than LLVM just to facilitate additional coalescing. For example, in function 65 Unison places two load instructions more than LLVM but in return it eliminates nine register-to-register move instructions.

ARM also benefits from Unison's aggressive coalescing, albeit to a lesser extent. Additional code size reduction for this processor is achieved by exploiting the Thumb-2 instruction set extension which allows 16- and 32-bit instructions to be freely mixed (see Section 4.3). For example, Unison reduces significantly the size of function 41 by selecting 14 more 16-bit instructions than LLVM, at the expense of an additional move instruction.

For Hexagon, LLVM's heuristic approach generates code of near-optimal size. Unison is only able to improve nine functions by coalescing aggressively (for example function 30), by reducing the amount of spilling (for example function 89), or by a combination of both (for example function 73).

Impact of integration. Unison's improvement over LLVM can be partially attributed to its integrated approach to register allocation and instruction scheduling. To evaluate the fundamental

Fig. 19. Code size reduction over LLVM (black) and gap (gray) for each function.

benefit of solving these problems in integration, we measure the improvement of the optimal solutions generated by Unison over those generated by solving register allocation and instruction scheduling optimally but in isolation. We call this optimal but isolated variant *Disunion*.

Disunion proceeds as follows. First, global register allocation is solved optimally according to the model in Section 5, assuming LLVM's given operation order. The objective function is similar to that of earlier combinatorial register allocation approaches (see Section 3). For speed optimization, the objective function minimizes total instruction latency weighted by execution frequency:

$$weight(b) = freq(b); \qquad \textbf{cost}(b) = \sum_{o \in O_b \,:\, \textbf{active}(o)} lat[\textbf{ins}(o)] \quad \forall b \in B. \tag{14}$$

For code size optimization, Unison's usual objective function (equation (7)) is employed. After global register allocation, Disunion solves instruction scheduling optimally according to the model in Section 6. The model is completed with additional dependencies caused by register assignment. The objective function for isolated instruction scheduling is the same as for Unison for both goals.

The mean improvement of solving register allocation and instruction scheduling in integration over the isolated approach ranges from a slight 0.7% to a significant 7.2% for the different goals and processors, as summarized in Table 4. These results confirm the benefit of integrating register allocation and instruction scheduling, although the extent depends on the goal and processor. In general, the integrated approach obtains better code through more effective spilling and coalescing, two program transformations that particularly benefit from simultaneous instruction scheduling.

Table 4. Mean improvement of Unison's optimal solutions over combinatorial approach in isolation.

goal	Hexagon	ARM	MIPS
speed	7.2%	0.7%	1.8%
code size optimization	2.4%	1.4%	2.2%

Additionally, the integrated approach generates shorter schedules for speed optimization, either by exploiting Hexagon's VLIW architecture or by hiding long latencies in the case of ARM and MIPS.

Comparing the improvement of Unison and Disunion over LLVM for the same subset of functions solved optimally reveals that the integration is a significant factor in the overall Unison improvement. For example, the mean speedup of Unison over LLVM for Hexagon on this function subset is 8.8%, while that of Disunion over LLVM is only 1%. In two cases, Disunion generates on average worse code than LLVM (2.2% mean code size increase for Hexagon and 0.3% mean slowdown for ARM). In these cases, the objectives of register allocation and instruction scheduling conflict to the extent that solving each problem optimally only decreases the quality of the generated code. Unison avoids this issue by solving register allocation and instruction scheduling simultaneously according to a single objective function.

The code quality benefits of Unison's integrated approach come at a price in terms of scalability compared to Disunion's decomposed approach. Given the same time limit, Disunion is able to solve optimally between 1.4 and 2.8 more functions than Unison, depending on the processor and goal.

11.3 Scalability

This section studies the scalability of Unison for the three evaluation processors and the same functions as in Section 11.2. The section aggregates speed and code size optimization results (unless otherwise stated), as the scalability trends are similar for both goals.

Overall scalability. The results indicate that Unison scales up to medium-sized functions, which puts 96% of all MediaBench and SPEC CPU2006 functions within reach. Figure 20 summarizes the solving complexity of the experiment functions. Figure 20a shows the solving time of each function solved optimally. Depending on the processor, Unison solves functions of up to 946 input instructions optimally in tens to hundreds of seconds. At the same time, Unison can time out for functions with as few as 30 input instructions. In such cases, high-quality solutions with moderate gaps are often obtained independently of the function size (mean gap 22.2%).

The gap information does not only provide insight into the quality of a solution, but can be used to push the scalability of our approach further while preserving its code quality guarantees. Figure 21 shows the accumulated percentage of acceptable solutions obtained over time for different gap requirements, when optimizing for speed on Hexagon. The solutions to 81% of the functions are proven optimal (0% gap) within the time limit. If the optimality requirement is relaxed to a certain gap limit, the scalability of Unison improves as more functions can be solved acceptably. For example, 90% of the functions are solved with less than 10% gap within the time limit. The percentage of acceptable solutions increases with diminishing returns for larger gap limits, as Figure 21 shows.

Impact of different processors. The scalability of Unison does not only depend on the function under compilation, but is also significantly affected by the targeted processor. The results show that the best scalability is achieved for Hexagon: Unison can solve optimally 2.9 and 3.5 times more medium-sized functions (between 100 and 1000 input instructions) for this processor than for ARM and MIPS. Furthermore, the largest function solved optimally for Hexagon has 946 input

(a) Solving time to optimality by function size.   (b) Accumulated % of optimal solutions over time.

Fig. 20. Solving complexity for different processors.

Fig. 21. Accumulated % of solutions over time for different gap requirements (speed optimization on Hexagon).

instructions, compared to 330 input instructions for ARM and 205 input instructions for MIPS (see Figure 20a). The same trend can be seen for small functions (up to 100 input instructions), where Unison times out in 73.2% and 26.8% of the cases for MIPS and ARM but not a single time for Hexagon. The median size of the functions for which Unison times out is 277, 220, and 184 input instructions for Hexagon, ARM, and MIPS. Given a fixed size, more functions are solved optimally and in less time for Hexagon while MIPS yields the fewest optimal solutions and the longest solving times (see Figure 20a). As a consequence, given a time limit, Unison solves more functions optimally for Hexagon, followed by ARM and MIPS as Figure 20b corroborates. Furthermore, for functions where Unison times out, the smallest and largest gaps are achieved for Hexagon and MIPS respectively, independently of function size.

The fact that Unison scales better for Hexagon (a VLIW processor) than for ARM and MIPS (single-issue processors) is surprising as the latter are considered easier to handle by heuristic approaches [55]. The fact that Unison scales better for ARM than for MIPS is also unexpected, as ARM includes more features for the solver to deal with such as selection of 16- and 32-bit instructions (see Section 4.3) and double-load and double-store instructions (see Section 9).

Table 5. Percentage of optimal solutions and mean gap with LLVM and trivial upper bounds.

processor	LLVM upper bound		trivial upper bound	
	optimal solutions	mean gap	optimal solutions	mean gap
Hexagon	79%	3.2%	61.5%	8.3%
ARM	62%	6.3%	58%	9.1%
MIPS	44%	14.5%	42%	20.1%

A factor that has been found to affect Unison's scalability differently depending on the targeted processor is the use of LLVM's solution cost as an upper bound on the objective function. Table 5 compares Unison's scalability (in percentage of optimal solutions and mean gap) using LLVM's and a trivial upper bound. With the LLVM upper bound (the default configuration), Unison clearly scales best for Hexagon, followed by ARM, and followed by MIPS. With the trivial upper bound, the results for Hexagon and ARM tend to even out, indicating that differences in LLVM's upper bound are the main cause of scalability divergence among these processors. In this alternative configuration, the relative difference between MIPS and these two processors persists, indicating that Unison is simply less effective in solving functions for MIPS.

A major factor that limits Unison's scalability for MIPS is the high overhead of modeling the preservation of its callee-saved registers. If these registers are not preserved, the scalability for MIPS improves significantly (1.6 times more functions are solved optimally), surpassing that for ARM and approaching that for Hexagon, which are only slightly affected. The reason for this difference is that MIPS requires 14 temporaries to hold the value of all callee-saved registers, whereas Hexagon and ARM only require six and three temporaries (by using double-store and double-load instructions for Hexagon and push and pop instructions for ARM). The number of such temporaries has a high impact on Unison's scalability as, after LSSA construction, it is multiplied by the number of basic blocks (as callee-saved temporaries are live through the entire function).

Another factor that also limits Unison's scalability for MIPS is the processor's wide range of instruction latencies (from one cycle for common arithmetic-logic instructions to 38 cycles for integer division). If a uniform, one-cycle latency model is assumed instead for all processors, the scalability for MIPS improves significantly (1.3 times more functions are solved optimally), approaching that of ARM which is less affected. The scalability for Hexagon remains unaltered, as this processor does not provide floating-point or costly integer arithmetic instructions.

A factor that has been evaluated and ruled out as a potential cause of divergence in Unison's scalability between Hexagon and the other two processors is the issue width. The scalability differences do not decrease if a single-issue pipeline is assumed for Hexagon. On the contrary, 1.1 times more functions are solved optimally for a single-issue version of Hexagon compared to the original VLIW architecture.

A more comprehensive evaluation of the effect of processor features and their interactions on Unison's scalability is part of future work.

Impact of solving methods. As is common in combinatorial optimization, the solver portfolio and solving improvements described in Section 10 are crucial for Unison's scalability.

The two solvers included in the portfolio (Section 10.1) are highly complementary: 49.7% of the solutions are delivered by the decomposition-based solver and the remaining ones by the off-the-shelf solver. The amount of solutions delivered by each solver is distributed rather evenly as function size increases, but the largest function solved by the decomposition-based solver (946 input instructions) is larger than the corresponding one by the off-the-shelf solver (874 input instructions).

(a) Solving time to optimality by function size.   (b) Accumulated % of optimal solutions over time.

Fig. 22. Solving complexity with and without solving improvements.

The better scalability of the decomposition-based solver is achieved by exploiting the structure of the LSSA form as explained in Section 10.1.

The solving improvements described in Section 10.2 allow Unison to find significantly more solutions, in particular for larger functions. Figure 22 summarizes the solving complexity of the experiment functions for all processors with and without solving improvements. The percentage of functions that are solved optimally grows from 44.7% to 61.7% when the solving improvements are applied. Many of the functions that are additionally solved are medium-sized: the solver with improvements can solve optimally 2.5 times more functions of this size than without improvements. The largest functions obtained with and without improvements have 946 and 647 input instructions respectively (see Figure 22a). The same trend can be observed for small functions, where Unison times out 2.1 times more often when the improvements are not applied. The median size of the functions for which Unison times out is of 218 input instructions with improvements and 184 input instructions without.

In general, when the improvements are enabled more functions are solved optimally and in less time for any size (see Figure 22a). Furthermore, given a time limit, significantly more functions are solved optimally (see Figure 22b). An exception is for the most trivial functions that are solved optimally within 0.3 seconds, for which the overhead introduced by the solving improvements is not amortized. Furthermore, for functions that cannot be solved optimally the improvements reduce the gap significantly: from 71.3% to 22.6% mean gap.

11.4   Actual Speedup

This section compares the *actual* (as opposed to *estimated*) execution speed of the code generated by Unison to that of LLVM. The section contributes the first actual speedup results for a combinatorial register allocation and instruction scheduling approach.

Processor, input functions, and execution platform. For the actual speedup experiments, we focus on Hexagon as the processor for which Unison improves most functions and estimates the highest speedup (see Figure 18). We select functions from MediaBench as this benchmark suite characterizes best the multimedia and communications applications targeted by Hexagon. MediaBench consists of 11 applications where most applications can be run in two modes (typically encoding and

Fig. 23. Actual function speedup over LLVM grouped by application and mode.
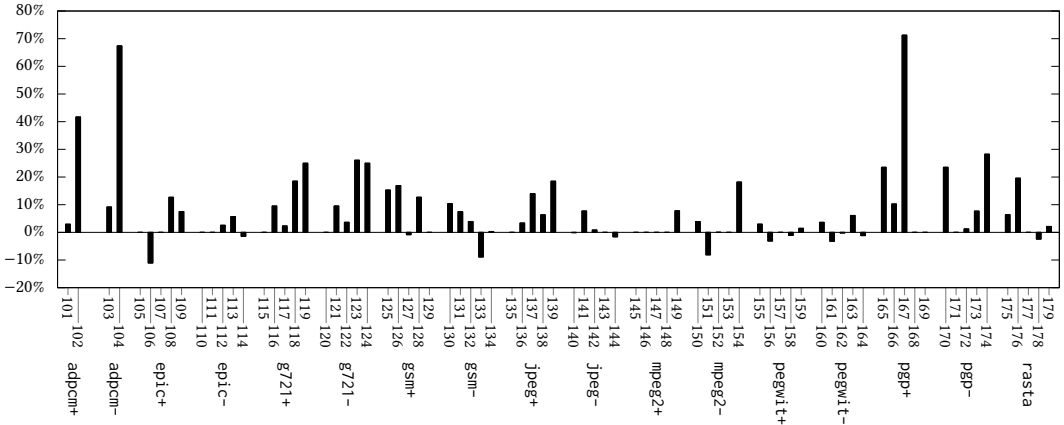
decoding, represented with + and − respectively). The applications ghostscript and mesa are excluded due to compilation errors in LLVM.

Each application and mode is compiled and profiled with LLVM using the reference input loads provided by MediaBench, and the top-five hottest functions (those that account for most execution time) are selected for compilation with Unison. This methodology is applied to ensure that the selected functions are sufficiently exercised during the execution and to enable a whole-application speedup comparison. The same methodology could be used in a practical setup to strike a balance between compilation time and code quality, exploiting the Pareto Principle which is commonly observed in software ("20% of the code accounts for 80% of the execution time"). In the case of MediaBench, the selected functions account for more than 66.7% of the execution time in all applications and modes except epic+, epic−, mpeg2+, and rasta where most execution time is spent in floating-point emulation library calls.

Appendix C details the features and percentage of the execution time of each selected function within its application and mode. The applications are executed on Qualcomm's Hexagon Simulator 6.4 [81]. Using a simulator eases reproducibility, provides detailed execution statistics, and permits experimenting with architectural variations (as in Appendix B) at the cost of a slight accuracy loss (cache simulation induces an error of up to 2% from perfect cycle-accuracy [81]). This paper assumes that this error is propagated similarly for LLVM and Unison's generated code.

Function speedup over LLVM. The results show that the speedup gained by Unison over LLVM for the hottest MediaBench functions is significant (6.6% mean speedup), despite a few individual slowdowns. Figure 23 shows the actual speedup over LLVM for each hot function in MediaBench, grouped by application and mode. For 60.8% of the functions, Unison speeds up the code generated by LLVM (up to 71.3%), while 17.7% of the functions are actually slowed down (down by 11.1%), contradicting Unison's estimation. Appendix B studies the factors behind this contradiction.

Application speedup over LLVM. The results show that the actual speedup demonstrated by Unison for individual functions propagates proportionally to whole applications (5.6% mean speedup). Figure 24 shows the actual speedup when Unison is applied to the five hottest functions of each application. Unison speeds up all applications (up to 15.8%), except pegwit+ which is slightly slowed down (−0.2%). epic+, epic−, mpeg2+, and rasta are excluded from the comparison as less than 5% of their execution time is spent on code generated by Unison. In general, Amdahl's

Fig. 24. Actual application speedup over LLVM.

law applies to the speedup results, as only a fraction of each application is compiled by Unison. Additional speedup can be expected if this fraction is increased.

12 CONCLUSION AND FUTURE WORK

This paper has introduced a combinatorial approach to integrated register allocation and instruction scheduling. It is the first approach of its kind to be *practical*, as it is *complete* (modeling all program transformations used in state-of-the art compilers), *scalable* (scaling to medium-sized functions of up to 1000 instructions), and *executable* (generating executable code). Key to satisfying these properties is the use of constraint programming to capture and exploit the structure underlying register allocation and instruction scheduling.

The approach is implemented in the open-source tool Unison. A thorough experimental evaluation on Hexagon, ARM, and MIPS confirms that Unison generates better code than LLVM (in terms of estimated speedup and code size reduction) while scaling to medium-sized functions. A significant part of this improvement is found to stem from the integrated nature of Unison. For the first time, the evaluation confirms that the speedup estimate for MediaBench benchmarks on Hexagon results in actual execution speedup.

Our approach can be used in practice to trade compilation time for code quality beyond the usual compiler optimization levels, fully exploit processor-specific features, and identify improvement opportunities in existing heuristic algorithms.

Future work. Future work includes addressing the model limitations discussed in Section 5 (lack of global support for multi-allocation) and Section 6 (local instruction scheduling). A first step towards global instruction scheduling could be to follow the superblock approach proposed by Malik *et al.* [66]. Additional improvements can also be expected from extending the scope of register allocation to multiple functions [97].

The actual speedup delivered by our approach can be improved by addressing the model inaccuracies that in the worst case can lead to slowdowns, as seen in Section 11.4 and discussed in Appendix B. This line of work includes characterizing those processor features responsible for the inaccuracy and capturing their effect in the cost model that underlies the objective function. A greater challenge is to devise accurate cost models for out-of-order processors such as x86 [52]. Supporting x86 in Unison is an ongoing effort [16].

Another potential direction is to improve scalability by exploring a broader array of solving methods and combinatorial optimization techniques. For example, hybrid integer and constraint

programming techniques tend to perform better than each of the techniques in isolation for a wide range of resource allocation and scheduling problems [64].

Finally, a longer-term goal is to integrate register allocation and instruction scheduling with instruction selection. Such an approach would not only deliver additional improvements, but also provide a framework for studying the trade-offs and interactions between different configurations of the three problems. A constraint-based approach to instruction selection is available together with a discussion of how the models could be integrated [51]. The main challenge lies in devising modeling and solving improvements to handle the considerable size of the resulting solution space.

ACKNOWLEDGMENTS

REFERENCES

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. 1993. Extending Chip in Order to Solve Complex Scheduling and Placement Problems. *Mathematical and Computer Modelling* 17, 7 (April 1993), 57–73.

[2] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN Notices* 33, 4 (April 1998), 17–20.

[3] Andrew W. Appel and Lal George. 2001. Optimal Spilling for CISC Machines with Few Registers. In *Programming Language Design and Implementation*. ACM, 243–253.

[4] ARM 2007. *ARM1156T2F-S Technical Reference Manual*. ARM. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0290g/DDI0290G_arm1156t2fs_r0p4_trm.pdf, accessed on 2019-06-19.

[5] John Aycock and R. Nigel Horspool. 2000. Simple Generation of Static Single-Assignment Form. In *Compiler Construction*, Vol. 1781. Springer-Verlag, 110–124.

[6] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. 2006. *Constraint-Based Scheduling and Planning*, Chapter 22, 759–797. In Rossi et al. [83].

[7] Gergö Barany and Andreas Krall. 2013. Optimal and Heuristic Global Code Motion for Minimal Spilling. In *Compiler Construction (LNCS)*, Vol. 7791. Springer-Verlag, 21–40.

[8] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. 2007. Optimal Bitwise Register Allocation Using Integer Linear Programming. In *Languages and Compilers for Parallel Computing*. Springer-Verlag, 267–282.

[9] Steven Bashford and Rainer Leupers. 1999. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems* 4 (March 1999), 119–165.

[10] Nicolas Beldiceanu and Evelyne Contejean. 1994. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling* 20, 12 (Dec. 1994), 97–123.

[11] Christian Bessiere. 2006. *Constraint Propagation*, Chapter 3, 27–81. In Rossi et al. [83].

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. IOS Press.

[13] Edward H. Bowman. 1959. The Schedule-Sequencing Problem. *Operations Research* 7, 5 (Sept. 1959), 621–624.

[14] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. In *Programming Language Design and Implementation*. ACM, 311–321.

[15] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. 2016. Verified Construction of Static Single Assignment Form. In *Compiler Construction*. ACM, 67–76.

[16] Mats Carlsson and Roberto Castañeda Lozano. 2018. Unison's source code: x86 fork. https://github.com/matsc-atsics-se/unison

[17] Roberto Castañeda Lozano. 2016. Tool Demonstration: Register Allocation and Instruction Scheduling in Unison. https://youtu.be/t4g2AjSfMX8

[18] Roberto Castañeda Lozano. 2017. Register Allocation and Instruction Scheduling in Unison. https://youtu.be/kx64V74Mba0

[19] Roberto Castañeda Lozano. 2017. The Unison Manual. https://unison-code.github.io/doc/manual.pdf

[20] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. 2012. Constraint-based register allocation and instruction scheduling. In *Principles and Practice of Constraint Programming (LNCS)*, Vol. 7514. Springer-Verlag, 750–766.

[21] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2014. Combinatorial Spill Code Optimization and Ultimate Coalescing. In *Languages, Compilers, Tools and Theory for Embedded Systems*. ACM,

23–32.

[22] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2016. Register Allocation and Instruction Scheduling in Unison. In *Compiler Construction*. ACM, 263–264.

[23] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2018. Unison website. https://unison-code.github.io

[24] Roberto Castañeda Lozano and Christian Schulte. 2014. *Survey on Combinatorial Register Allocation and Instruction Scheduling*. Technical Report. SCALE, KTH Royal Institute of Technology & Swedish Institute of Computer Science. Archived at arXiv:1409.7628 [cs.PL]. A revised and extended version has been accepted for publication at *ACM Computing Surveys*.

[25] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6, 1 (1981), 47–57.

[26] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. 1997. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications* 34 (Nov. 1997), 1–14. Issue 9.

[27] Frederick Chow. 1988. Minimizing Register Usage Penalty at Procedure Calls. In *Programming Language Design and Implementation*. ACM, 85–94.

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

[29] Geoffrey G. Chu. 2011. *Improving combinatorial optimization*. Ph.D. Dissertation. The University of Melbourne, Australia.

[30] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. 2014. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro* 34, 2 (March 2014), 34–43.

[31] Quentin Colombet, Florian Brandner, and Alain Darte. 2015. Studying Optimal Spilling in the Light of SSA. *ACM Transactions on Architecture and Code Optimization* 11, 4 (Jan. 2015), 1–26.

[32] Keith Cooper and Linda Torczon. 2012. *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). MIT Press.

[34] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490.

[35] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. 2009. Progressive Spill Code Placement. In *Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 77–86.

[36] Tobias J.K. Edler von Koch, Igor Böhm, and Björn Franke. 2010. Integrated Instruction Selection and Register Allocation for Compact Code Generation Exploiting Freeform Mixing of 16- and 32-bit Instructions. In *Code Generation and Optimization*. ACM, 180–189.

[37] Mattias Eriksson and Christoph W. Kessler. 2012. Integrated Code Generation for Loops. *ACM Transactions on Embedded Computing Systems* 11S, 1 (June 2012), 1–24.

[38] Heiko Falk, Norman Schmitz, and Florian Schmoll. 2011. WCET-aware Register Allocation Based on Integer-Linear Programming. In *Euromicro Conference on Real-Time Systems*. IEEE, 13–22.

[39] Joseph A. Fisher. 1983. Very Long Instruction Word Architectures and the ELI-512. In *International Symposium on Computer Architecture*. ACM, 140–150.

[40] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2015. Horn clauses as an intermediate representation for program analysis and transformation. In *Theory and Practice of Logic Programming*. Cambridge University Press, 526–542.

[41] Catherine H. Gebotys. 1997. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *System Synthesis*. IEEE, 41–47.

[42] Gecode Team. 2018. Gecode: Generic Constraint Development Environment. https://www.gecode.org

[43] Carla P. Gomes and Bart Selman. 2001. Algorithm portfolios. *Artificial Intelligence* 126, 1 (Feb. 2001), 43 – 62.

[44] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. 2000. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning* 24, 1-2 (Feb. 2000), 67–100.

[45] James R. Goodman and Wei-Chung Hsu. 1988. Code Scheduling and Register Allocation in Large Basic Blocks. In *International Conference on Supercomputing*. ACM, 442–452.

[46] David W. Goodwin and Kent Wilken. 1996. Optimal and near-optimal global register allocations using 0-1 integer programming. *Software: Practice and Experience* 26 (Aug. 1996), 929–965. Issue 8.

[47] R. Govindarajan. 2007. Instruction Scheduling. In *The Compiler Design Handbook* (2nd ed.). CRC.

[48] Sebastian Hack, Daniel Grund, and Gerhard Goos. 2006. Register Allocation for Programs in SSA-Form. In *Compiler Construction (LNCS)*, Vol. 3923. Springer-Verlag, 247–262.

[49] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.

[50] Gabriel Hjort Blindell. 2016. *Instruction Selection: Principles, Methods, and Applications*. Springer-Verlag.

[51] Gabriel Hjort Blindell. 2018. *Universal Instruction Selection*. Doctoral dissertation. KTH Royal Institute of Technology, Sweden.

[52] Intel 2019. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Intel. https://software.intel.com/en-us/articles/intel-sdm, accessed on 2019-06-19.

[53] Cliff Young Joseph A. Fisher, Paolo Faraboschi. 2005. *Embedded Computing*. Elsevier.

[54] Daniel Kästner. 2001. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Languages, Compilers, Tools and Theory for Embedded Systems (LNCS)*, Vol. 1985. Springer-Verlag, 63–80.

[55] Christoph W. Kessler. 2010. Compiling for VLIW DSPs. In *Handbook of Signal Processing Systems*. Springer-Verlag, 603–638.

[56] Christoph W. Kessler and Andrzej Bednarski. 2006. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience* 18 (2006), 1353–1390. Issue 11.

[57] David Ryan Koes and Seth Copen Goldstein. 2006. A Global Progressive Register Allocator. In *Programming Language Design and Implementation*. ACM, 204–215.

[58] David Ryan Koes and Seth Copen Goldstein. 2009. Register Allocation Deconstructed. In *Software and Compilers for Embedded Systems*. ACM, 21–30.

[59] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization*. IEEE, 75–88.

[60] Yat Chiu Law and Jimmy H. M. Lee. 2004. Global Constraints for Integer and Set Value Precedence. In *Principles and Practice of Constraint Programming (LNCS)*, Vol. 3258. Springer-Verlag, 362–376.

[61] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *International Symposium on Microarchitecture*. IEEE, 330–335.

[62] Rainer Leupers and Peter Marwedel. 2001. *Retargetable Compiler Technology for Embedded Systems: Tools and Applications*. Springer-Verlag.

[63] Andrea Lodi, Silvano Martello, and Michele Monaci. 2002. Two-dimensional packing problems: A survey. *European Journal of Operational Research* 141, 2 (Sept. 2002), 241 – 252.

[64] Michele Lombardi and Michela Milano. 2012. Optimal methods for resource allocation and scheduling: a cross-disciplinary survey. *Constraints* 17, 1 (Jan. 2012), 51–85.

[65] James B. MacQueen. 1967. Some Methods for Classification and Analysis of MultiVariate Observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 281–297.

[66] Abid M. Malik, Michael Chase, Tyrel Russell, and Peter Van Beek. 2008. An Application of Constraint Programming to Superblock Instruction Scheduling. In *Principles and Practice of Constraint Programming (LNCS)*, Vol. 5202. Springer-Verlag, 97–111.

[67] Abid M. Malik, Jim McInnes, and Peter Van Beek. 2008. Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors Using Constraint Programming. *Artificial Intelligence Tools* 17, 1 (2008), 37–54.

[68] Alan S. Manne. 1960. On the Job-Shop Scheduling Problem. *Operations Research* 8, 2 (March 1960), 219–223.

[69] MiniZinc Team. 2018. MiniZinc constraint modeling language. https://www.minizinc.org

[70] MIPS 2016. *The MIPS32 Instruction Set Manual*. MIPS. https://www.mips.com/downloads/the-mips32-instruction-set-v6-05/, accessed on 2019-06-19.

[71] Santosh G. Nagarakatte and R. Govindarajan. 2007. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *Compiler Construction (LNCS)*, Vol. 4420. Springer-Verlag, 126–140.

[72] Mayur Naik and Jens Palsberg. 2002. Compiling with Code-size Constraints. In *Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 120–129.

[73] V. Krishna Nandivada and Jens Palsberg. 2006. SARA: Combining Stack Allocation and Register Allocation. In *Compiler Construction (LNCS)*, Vol. 3923. Springer-Verlag, 232–246.

[74] V. Krishna Nandivada, Fernando Pereira, and Jens Palsberg. 2007. A Framework for End-to-End Verification and Evaluation of Register Allocators. In *Static Analysis (LNCS)*, Vol. 4634. Springer-Verlag, 153–169.

[75] George L. Nemhauser and Laurence A. Wolsey. 1999. *Integer and Combinatorial Optimization*. Wiley.

[76] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *CP (LNCS)*, Vol. 4741. Springer-Verlag, 529–543.

[77] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register allocation by puzzle solving. In *Programming Language Design and Implementation*. ACM, 216–226.

[78] Martin Persson. 2017. *Evaluating Unison's Speedup Estimation.* Master's thesis. KTH Royal Institute of Technology, Sweden.

[79] Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy K. John. 2005. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. In *International Symposium on Performance Analysis of Systems and Software*. IEEE, 10–20.

[80] Qualcomm 2013. *Hexagon Application Binary Interface Specification.* Qualcomm. https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools, accessed on 2019-06-19.

[81] Qualcomm 2013. *Hexagon Simulator User Guide.* Qualcomm. https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools, accessed on 2019-06-19.

[82] Bantwal Ramakrishna Rau and Joseph A. Fisher. 1993. Instruction-level parallel processing: history, overview, and perspective. *Journal of Supercomputing* 7 (May 1993), 9–50. Issue 1-2.

[83] Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). 2006. *Handbook of Constraint Programming.* Elsevier.

[84] Martin Savelsbergh. 1994. Preprocessing and Probing Techniques for Mixed Integer Programming Problems. *ORSA Journal on Computing* 6, 4 (Sept. 1994), 445–454.

[85] Bernhard Scholz and Erik Eckstein. 2002. Register Allocation for Irregular Architectures. In *Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 139–148.

[86] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach. *ACM Transactions on Architecture and Code Optimization* 10, 3 (Sept. 2013), 1–31.

[87] Helmut Simonis and Barry O'Sullivan. 2008. Search Strategies for Rectangle Packing. In *Principles and Practice of Constraint Programming (LNCS)*, Vol. 5202. Springer-Verlag, 52–66.

[88] Barbara M. Smith. 2006. *Modelling*, Chapter 11, 375–404. In Rossi et al. [83].

[89] Charles Spearman. 1904. The proof and measurement of association between two things. *The American Journal of Psychology* 15, 1 (Jan. 1904), 72–101.

[90] SPEC 2016. *CPU 2006 Benchmarks.* SPEC. https://www.spec.org/cpu2006, accessed on 2016-08-26.

[91] SPEC 2018. *Building the SPEC CPU2006 Tool Suite.* SPEC. https://www.spec.org/cpu2006/Docs/tools-build.html, accessed on 2018-03-21.

[92] Vugranam Sreedhar, Roy Ju, David Gillies, and Vatsa Santhanam. 1999. Translating Out of Static Single Assignment Form. In *Static Analysis (LNCS)*, Vol. 1694. Springer-Verlag, 849–849.

[93] Peter van Beek. 2006. *Backtracking Search Algorithms*, Chapter 4, 83–132. In Rossi et al. [83].

[94] Pascal Van Hentenryck and Jean-Philippe Carillon. 1988. Generality vs. Specificity: an Experience with AI and OR Techniques. In *National Conference on Artificial Intelligence*. AAAI Press, 660–664.

[95] Willem-Jan van Hoeve and Irit Katriel. 2006. *Global Constraints*, Chapter 7, 205–244. In Rossi et al. [83].

[96] Harvey M. Wagner. 1959. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly* 6, 2 (June 1959), 131–140.

[97] David W. Wall. 1986. Global Register Allocation at Link Time. In *SIGPLAN Symposium on Compiler Construction*. ACM, 264–275.

[98] Fredrik Wickberg and Mattias Eriksson. 2017. Outperforming state-of-the-art compilers in Unison. Ericsson research blog entry, https://www.ericsson.com/research-blog/outperforming-state-art-compilers-unison.

[99] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (Dec. 1945), 80–83.

[100] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal Instruction Scheduling Using Integer Programming. In *Programming Language Design and Implementation*. ACM, 121–133.

[101] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. 1994. An integrated approach to retargetable code generation. In *High-level Synthesis*. IEEE, 70–75.

[102] Tom Wilson, Gary Grewal, Shawn Henshall, and Dilip Banerji. 2002. An ILP-Based Approach to Code Generation. In *Code Generation for Embedded Processors*. Springer-Verlag, 103–118.

[103] Sebastian Winkel. 2007. Optimal Versus Heuristic Global Code Scheduling. In *International Symposium on Microarchitecture*. IEEE, 43–55.

Table 6. Parameters of global register allocation and instruction scheduling (without extensions).

Program parameters:

B, O, P, T	sets of basic blocks, operations, operands and temporaries
O_b, T_b	sets of operations and temporaries in basic block b
$operation(p)$	operation to which operand p belongs
$definer(t)$	operand that potentially defines temporary t
$users(t)$	operands that potentially use temporary t
$copy(o)$	whether operation o is a copy operation
$width(t)$	number of register atoms that temporary t occupies
$p \triangleright r$	whether operand p is pre-assigned to register r
$p \equiv q$	whether operands p and q are congruent

Processor parameters:

R, S	sets of registers and resources
$instrs(o)$	set of instructions that can implement operation o
$class(p, i)$	register class of operand p when implemented by instruction i
$lat(i)$	latency of instruction i
$con(i, s)$	consumption of processor resource s by instruction i
$dur(i, s)$	duration of usage of processor resource s by instruction i
$cap(s)$	capacity of processor resource s

Objective function parameters:

$weight(b)$	weight of basic block b
$\mathbf{cost}(b)$	cost of basic block b

A   COMPLETE MODEL

This appendix summarizes the complete model as introduced in Sections 4-8, excluding the additional program transformations from Section 9. Table 6 lists all parameters for the input program, the processor, and the objective function, whereas Table 7 lists all variables and constraints.

B   ACCURACY OF THE SPEEDUP ESTIMATION

This appendix studies the accuracy of Unison's speedup estimation which is based on equations (5) and (6) by comparing to the actual speed measurements for MediaBench functions on Hexagon. The appendix extends the methodology and results from a prestudy by Persson [78].

Accuracy. Figure 25 (*original processor* points) depicts the relation between estimated and actual speedup for each of the MediaBench functions studied in Section 11.4. As the figure shows, Unison's speedup estimation suffers from inaccuracies. This is confirmed by a Wilcoxon signed-rank test [99] which is used as an alternative to the more common t-test since the speedup difference (estimated minus actual speedup) does not follow a normal distribution. A significance level of 1% is required for all statistical tests. The estimation error (the absolute value of the speedup difference) follows an exponential distribution where the error is less than 1% for 36.7% of the functions, between 1% and 10% for 50.6% of the functions, and greater than 10% (up to 20%) for 12.7% of the functions. Unison's speedup estimation is biased in that it overestimates the actual speedup for 71% of the functions where there is an estimation error.

Table 7. Combinatorial model of global register allocation and instruction scheduling (without extensions).

Variables:

$\mathbf{reg}(t) \in R$	register to which temporary t is assigned	(V1)
$\mathbf{ins}(o) \in instrs(o)$	instruction that implements operation o	(V2)
$\mathbf{temp}(p) \in temps(p)$	temporary used or defined by operand p	(V3)
$\mathbf{live}(t) \in \mathbb{B}$	whether temporary t is live	(V4)
$\mathbf{active}(o) \in \mathbb{B}$	whether operation o is active	(V5)
$\mathbf{issue}(o) \in \mathbb{N}_0$	issue cycle of operation o from the beginning of its basic block	(V6)
$\mathbf{start}(t), \mathbf{end}(t) \in \mathbb{N}_0$	live start and end cycles of temporary t	(V7)

Register allocation constraints:

$$\text{no-overlap}\left(\{\langle \mathbf{reg}(t), \mathbf{reg}(t) + width(t), \mathbf{start}(t), \mathbf{end}(t)\rangle : t \in T_b \wedge \mathbf{live}(t)\}\right) \quad \forall b \in B \quad \text{(C1.1)}$$

$$\mathbf{reg}[\mathbf{temp}(p)] = r \quad \forall p \in P : p \triangleright r \quad \text{(C2.1)}$$

$$\mathbf{reg}[\mathbf{temp}(p)] \in class[p, \mathbf{ins}(operation(p))] \quad \forall p \in P : \mathbf{active}(operation(p)) \quad \text{(C3.2)}$$

$$\mathbf{active}(o) \quad \forall o \in O : \neg copy(o) \quad \text{(C4)}$$

$$\begin{aligned} \mathbf{live}(t) &\iff \mathbf{active}(operation(definer(t))) \\ &\iff \exists p \in users(t) : \mathbf{active}(operation(p)) \wedge \mathbf{temp}(p) = t \quad \forall t \in T \end{aligned} \quad \text{(C5)}$$

$$\mathbf{reg}[\mathbf{temp}(p)] = \mathbf{reg}[\mathbf{temp}(q)] \quad \forall p, q \in P : p \equiv q \quad \text{(C6)}$$

Instruction scheduling constraints:

$$\begin{aligned} \mathbf{issue}(operation(q)) &\geq \mathbf{issue}(operation(p)) + lat[\mathbf{ins}(operation(p))] \\ &\forall t \in T, \forall p \in \{definer(t)\}, \forall q \in users(t) : \mathbf{active}(operation(q)) \wedge \mathbf{temp}(q) = t \end{aligned} \quad \text{(C7.1)}$$

$$\begin{aligned} \text{cumulative}\left(\{\langle \mathbf{issue}(o), dur[\mathbf{ins}(o), s], con[\mathbf{ins}(o), s]\rangle : o \in O_b \wedge \mathbf{active}(o)\}, cap(s)\right) \\ \forall b \in B, \forall s \in S \end{aligned} \quad \text{(C8.1)}$$

Integration constraints:

$$\mathbf{start}(t) = \mathbf{issue}(operation(definer(t))) \quad \forall t \in T : \mathbf{live}(t) \quad \text{(C9)}$$

$$\mathbf{end}(t) = \max_{p \in users(t) : \mathbf{temp}(p) = t} \mathbf{issue}(operation(p)) \quad \forall t \in T : \mathbf{live}(t) \quad \text{(C10)}$$

Objective function:

$$minimize \sum_{b \in B} weight(b) \times \mathbf{cost}(b) \quad \text{((5))}$$

Despite its inaccuracy, the estimation is monotonically related to the actual speedup, in that higher estimated speedup often leads to higher actual speedup. A Spearman correlation coefficient [89] of 0.7 confirms that the monotonic relationship is strong (0 indicates no relationship and 1 indicates a perfect monotonically increasing relationship). Similarly to the Wilcoxon test, this measure is

Fig. 25. Relation between estimated and actual speedup over LLVM.

used as an alternative to the more common Pearson correlation coefficient because the speedup difference is not normally distributed.

Accuracy factors. The potential sources of inaccuracy in a static speedup estimation are:

Late program transformations. Late compiler, assembler, and link-time optimizations can modify the program under compilation after Unison and invalidate the speedup estimation.

Dynamic processor behavior. Unpredictable processor features such as cache memories and branch prediction are not captured by Unison's speedup estimation and can lead to processor stalls that are unaccounted for.

Dynamic program behavior. The estimated execution frequency of each basic block (equation (6)) might deviate from the actual execution frequency.

The first source (late program transformations) is controlled in the experiments by simply disabling all optimizations that run after Unison (see Table 3). The influence of the other sources is determined by measuring the actual speedup on an *ideal* Hexagon processor without pipeline stalls. Estimation errors for such a processor are solely due to the execution frequency estimation provided by LLVM. The study of the effect of enabling late program transformations in the speedup estimation's accuracy is part of future work.

The results show that the dynamic processor behavior is the main source of inaccuracy in Unison's speedup estimation and the sole responsible for overestimation. The *ideal processor* points in Figure 25 depict the relation between estimated and actual speedup on a Hexagon processor without stalls, where the dynamic program behavior is the sole source of inaccuracy in the estimated speedup. For this configuration, the percentage of the functions that are slowed down drops to 3.8%, the percentage of functions for which the estimation error is less than 1% increases to 60.8%, and the Spearman correlation coefficient increases to a very strong 0.89. Furthermore, the overestimation bias in the speedup estimation is dampened as seen in Figure 25 (*ideal processor* points).

Implications. Unison's estimated speedup is generally greater than the actual speedup, but both are strongly monotonically related. This fact is key to the combinatorial approach as it motivates investing additional computational effort in finding solutions of increasing quality. In practical terms, for MediaBench functions on Hexagon the actual speedup can be expected to be lower than estimated, but solutions of increasing estimated speedup can be expected to indeed yield higher speedup. The accuracy of the speedup estimation can be expected to improve for more predictable processors than Hexagon (for example, with scratchpad rather than cache memories) and vice versa.

Table 8. MediaBench and SPEC CPU2006 functions for the evaluation of estimated code quality and scalability.

id	name (app)	I	I/B	RP	ILP	CR	CS	id	name (app)	I	I/B	RP	ILP	CR	CS
1	handle_noinline_attr.. (gcc)	28	7	0	2.2	3.6	44	51	autohelperowl_atta.. (gobmk)	53	14	0	2.1	3.8	110
2	control_flow_insn_p (gcc)	78	7	0	2	5.1	144	52	autohelperowl_defe.. (gobmk)	34	34	0	2.4	8.9	108
3	insert_insn_on_edge (gcc)	55	7	0	1.5	11.1	49	53	autohelperowl_defe.. (gobmk)	30	8	0	2.3	6.7	180
4	update_br_prob_note (gcc)	49	5	0	1.8	4.1	145	54	autohelperpat1114 (gobmk)	42	12	0	1.9	4.8	140
5	_cpp_init_internal_p.. (gcc)	49	49	0	2.1	10.4	42	55	autohelperpat335 (gobmk)	30	30	0	2.7	3.4	178
6	lex_macro_node (gcc)	77	7	0	1.8	6.5	172	56	autohelperpat508 (gobmk)	27	27	0	2.4	3.7	142
7	cse_basic_block (gcc)	779	5	2.9	1.8	4	62	57	autohelperpat83 (gobmk)	53	16	0	2.4	3.8	148
8	rtx_equal_for_cselib_p (gcc)	299	5	1.6	1.6	3	83	58	simple_showboard (gobmk)	208	8	2.9	1.8	5.8	68
9	debug_df_chain (gcc)	49	12	0	2.2	8.3	146	59	skip_intrabk_SAD (h264ref)	318	11	0	1.6	0	37
10	modified_type_die (gcc)	743	7	0.4	1.7	5.1	85	60	free_orig_planes (h264ref)	74	12	0	1.6	9.7	62
11	emit_note (gcc)	87	6	0	1.4	2.3	85	61	GetSkipCostMB (h264ref)	248	16	11.1	1.9	2.3	31
12	gen_sequence (gcc)	70	6	0	1.9	2.9	125	62	writeSyntaxEleme.. (h264ref)	99	8	0	2	0	76
13	subreg_hard_regno (gcc)	88	8	0	1.6	7.7	108	63	GSIAddKeyToIndex (hmmer)	88	12	0	1.7	5.8	75
14	split_double (gcc)	142	8	0	1.7	5	125	64	EVDBasicFit (hmmer)	229	17	0.2	2	5.1	42
15	add_to_mem_set_list (gcc)	59	6	0	1.6	3.4	142	65	SampleDirichlet (hmmer)	99	10	0	2.1	8.5	81
16	find_regno_partial (gcc)	52	4	0	1.4	0	91	66	DegenerateSymbolSc.. (hmmer)	98	11	0.1	2.1	4.7	64
17	use_return_register (gcc)	72	6	0	1.4	5.7	95	67	Plan7SetCtime (hmmer)	38	12	0	1.6	13.6	40
18	ix86_expand_move (gcc)	347	6	0	1.6	4.9	89	68	MSAToSqinfo (hmmer)	229	8	0.5	1.6	5.7	60
19	legitimate_pic_addre.. (gcc)	208	5	0.1	1.4	1	81	69	null_convert (jpeg)	110	6	0.3	2.7	0	29
20	gen_extendsfdf2 (gcc)	62	15	0	1.9	11.7	93	70	jinit_c_prep_contro.. (jpeg)	193	13	3.3	2.3	3.8	49
21	gen_mulsidi3 (gcc)	76	75	0	2.9	10.7	87	71	glFogf (mesa)	38	8	0	1.6	7.9	151
22	gen_peephole2_1255 (gcc)	83	82	0	2.5	12.4	66	72	glNormal3d (mesa)	64	59	0.5	2.4	5.5	51
23	gen_peephole2_1271 (gcc)	102	101	0	2.3	13.1	68	73	glRasterPos3d (mesa)	52	10	0	2	7.7	146
24	gen_peephole2_1277 (gcc)	153	51	0	2.4	12	51	74	glTexCoord2d (mesa)	24	24	0	2.4	6.7	97
25	gen_pfnacc (gcc)	145	144	0	2.7	11.2	55	75	gl_stippled_bresenham (mesa)	223	11	6.5	2.2	1.3	59
26	gen_rotlsi3 (gcc)	32	31	0	2.3	13	120	76	gl_save_Frustum (mesa)	109	10	0.2	2.1	5.4	110
27	gen_split_1001 (gcc)	264	29	0	2	11.5	33	77	gl_save_LineWidth (mesa)	78	7	0	1.8	5.2	114
28	gen_split_1028 (gcc)	271	269	0	2.5	12.7	17	78	translate_id (mesa)	68	5	0	1.8	0.6	93
29	gen_sse_nandti3 (gcc)	33	32	0	3	9.5	126	79	gl_Map1f (mesa)	341	6	0.1	1.9	5	41
30	gen_sunge (gcc)	34	10	0	2.1	15.2	39	80	smooth_ci_line (mesa)	186	12	2.5	2.1	3.1	60
31	insert_loop_mem (gcc)	117	6	0	1.7	1.7	133	81	free_unified_knots (mesa)	39	4	0	1.5	10.5	77
32	eiremain (gcc)	236	9	0.5	1.9	2.7	88	82	tess_test_polygon (mesa)	641	7	1.3	1.9	4.7	31
33	elimination_effects (gcc)	522	5	1.3	1.7	1.6	69	83	auxWireBox (mesa)	122	12	0.1	2.6	8.4	26
34	gen_reload (gcc)	503	9	0.2	1.7	7.4	93	84	gl_ColorPointer (mesa)	94	6	0	1.5	3.3	45
35	reload_cse_simplify_.. (gcc)	222	7	1.7	1.7	5.9	80	85	r_serial (milc)	536	15	4.1	2	7.3	34
36	simplify_binary_is2o.. (gcc)	62	62	0.8	2.1	3.2	18	86	scalar_mult_sub_su3.. (milc)	156	137	0	2.6	4.2	14
37	remove_phi_alternative (gcc)	43	5	0	1.7	0	145	87	Decode_MPEG1_Non_I.. (mpeg2)	238	6	3.7	1.7	3.8	64
38	contains_placeholder_p (gcc)	202	4	0.1	1.5	4.5	93	88	cpDecodeSecret (pegwit)	23	23	0	2.6	13.3	31
39	assemble_end_function (gcc)	179	10	0	1.7	8.5	116	89	vlShortLshift (pegwit)	83	6	0	2.3	1.3	41
40	default_named_sectio.. (gcc)	72	13	0	1.9	8.4	155	90	encryptfile (pgp)	411	13	0.8	1.9	7.4	46
41	sample_unpac.. (ghostscript)	98	7	0	2.1	0	61	91	make_canonical (pgp)	79	18	0	2	11.6	75
42	autohelperattpat10 (gobmk)	46	11	0.3	2	6.6	175	92	LANG (pgp)	658	10	1.4	1.8	7.7	28
43	autohelperbarriers.. (gobmk)	111	15	2	2.2	4.5	54	93	MD5Transform (pgp)	551	546	4.2	1.7	0	5
44	atari_atari_attack.. (gobmk)	515	6	3.3	1.7	3.8	73	94	mp_display (pgp)	255	13	0.8	2	9.2	55
45	compute_aa_status (gobmk)	206	5	4.3	1.6	2.9	84	95	comp_Jboundaries (rasta)	45	9	0	1.8	5.1	71
46	dragon_weak (gobmk)	63	9	0	1.7	2.7	124	96	is_draw (sjeng)	92	6	0	1.8	0	64
47	get_saved_worms (gobmk)	135	7	6.1	1.7	3.1	46	97	push_king (sjeng)	103	9	0	1.4	0	29
48	read_eye (gobmk)	170	7	2.3	1.8	2.6	95	98	stat_retry (sphinx3)	70	9	0	1.8	8.6	81
49	topological_eye (gobmk)	465	7	7.1	1.9	2.3	54	99	lextree_subtree_.. (sphinx3)	146	11	0	2.2	7.6	63
50	autohelperowl_atta.. (gobmk)	61	12	0	2.2	4.9	218	100	lm_tg_score (sphinx3)	261	7	0.2	1.7	3.5	45

For applications exhibiting more irregular control behavior than those present in MediaBench, the speedup estimation can be improved by profile-guided optimization.

C FUNCTIONS

Table 8 lists the MediaBench and SPEC CPU2006 functions for the evaluation of estimated code quality (Section 11.2) and scalability (Section 11.3). Their features are: input instructions (**I**), input instructions per basic block (**I/B**), register pressure (**RP**), instruction-level parallelism (**ILP**), call instruction ratio (**CR**), and cluster size (**CS**) in the sampling algorithm. All features are averaged over the three studied processors.

Table 9 lists the MediaBench functions for the evaluation of actual speedup (Section 11.4) and accuracy of the speedup estimation (Appendix B) on Hexagon V4. Their features are: input instructions (**I**), input instructions per basic block (**I/B**), register pressure (**RP**), instruction-level parallelism (**ILP**), call instruction ratio (**CR**), and percentage of the execution time (**EX**).

Table 9. MediaBench functions for the evaluation of actual speedup and accuracy of the speedup estimation on Hexagon V4.

id	name	I	I/B	RP	ILP	CR	EX	id	name	I	I/B	RP	ILP	CR	EX
adpcm+								jpeg-							
101	adpcm_coder	87	8	0	2.1	0	96.9	140	ycc_rgb_convert	158	14	0	1.7	0	29.3
102	main	83	14	0	1.7	10.8	0.1	141	jpeg_idct_islow	235	21	0.9	2.1	0	28.9
adpcm-								142	h2v2_fancy_upsample	384	16	0	2.3	0	12.5
103	adpcm_decoder	66	11	0	1.9	0	95.2	143	decode_mcu	413	8	1.4	1.8	2.4	12.1
104	main	75	12	0	1.6	10.7	0.1	144	jpeg_fill_bit_buffer	118	6	0	1.5	2.5	3.6
epic+								mpeg2+							
105	quantize_image	751	40	0.5	2.3	11.9	0.2	145	fdct	772	86	1	2.5	14.2	0.8
106	run_length_encode_zeros	123	7	0.8	1.8	0.8	0.1	146	fullsearch	192	8	6.2	1.8	1.6	0.4
107	encode_stream	67	5	0	1.7	3	0.1	147	dist1	337	13	0	3	0	0.2
108	ReadMatrixFromPGMStream	191	11	0	1.9	8.9	0.1	148	putbits	149	8	0	2.2	3.4	0.2
109	main	775	60	0	2.2	11.5	0	149	calcSNR1	552	61	0.8	2.5	14.3	0.1
epic-								mpeg2-							
110	main	567	24	0.1	1.9	10.1	1.6	150	conv420to422	260	22	3.2	2.7	0	20.1
111	unquantize_image	301	9	0	1.7	11.3	1.1	151	form_component_prediction	313	6	0	2.2	0	13.8
112	read_and_huffman_decode	236	8	0	1.7	2.1	0.8	152	putbyte	20	7	0	1.5	5	13.7
113	write_pgm_image	130	13	0	2.1	6.9	0.6	153	Add_Block	288	16	0	1.9	0	10.2
114	internal_int_transpose	203	5	0	2	4.4	0.5	154	idctcol	103	26	0	1.7	0	8.9
g721+								pegwit+							
115	quan	21	4	0	1.8	0	46.4	155	gfAddMul	179	6	0	1.5	0.6	42.6
116	fmult	54	18	0	2.1	1.9	19.3	156	gfMultiply	334	6	0	1.6	2.1	29.7
117	update	383	7	0.5	2	0.8	3.2	157	squareEncrypt	451	6	0	2.6	0	6.1
118	g721_encoder	94	16	0	1.9	8.5	3.2	158	gfInvert	253	11	0	1.8	8.3	3.2
119	predictor_zero	61	61	0	2.3	9.8	2	159	gfSquare	191	5	0	1.5	2.1	2.5
g721-								pegwit-							
120	quan	21	4	0	1.8	0	44.5	160	gfAddMul	179	6	0	1.5	0.6	41.1
121	fmult	54	18	0	2.1	1.9	19.4	161	gfMultiply	334	6	0	1.6	2.1	28.6
122	update	383	7	0.5	2	0.8	11.7	162	squareDecrypt	451	6	0	2.6	0	11.4
123	g721_decoder	87	14	0	1.8	8	2.8	163	gfInit	143	11	0	2	2.1	4.2
124	predictor_zero	61	61	0	2.3	9.8	2	164	gfInvert	253	11	0	1.8	8.3	3.1
gsm+								pgp+							
125	Calculation_of_the_LTP_pa..	504	14	5.5	2	1.8	40.8	165	mp_smul	30	8	0	1.7	0	50.6
126	Short_term_analysis_filte..	121	20	0	2.6	0	25.3	166	longest_match	80	5	0	1.5	0	13.2
127	Gsm_Preprocess	107	7	1.2	1.6	2.8	8.8	167	fill_window	255	23	0	2.5	0.8	4.3
128	Weighting_filter	61	20	0	2.1	0	3.9	168	deflate	331	11	2.2	2.2	2.4	3.7
129	Autocorrelation	675	22	0.2	2.5	0.6	3.8	169	mp_compare	31	6	0	2	0	3.5
gsm-								pgp-							
130	Short_term_synthesis_filt..	109	6	0	1.7	0	73.7	170	mp_smul	30	8	0	1.7	0	66.2
131	Gsm_Long_Term_Synthesis_F..	158	14	0	1.9	1.3	6	171	mp_compare	31	6	0	2	0	4.6
132	Postprocessing	119	40	0	2.2	0	5.7	172	ideaCipher	204	6	0	1.8	0	4
133	APCM_inverse_quantization	92	10	0	1.7	7.6	2.5	173	mp_quo_digit	30	30	0	1.9	0	2.8
134	gsm_asr	22	3	0	1.8	0	1	174	MD5Transform	484	484	8	1.7	0	2.5
jpeg+								rasta							
135	forward_DCT	287	9	0	1.7	3.1	22.5	175	audspec	330	12	0.1	2.7	7.3	0.1
136	rgb_ycc_convert	146	15	0	2.1	0	21.1	176	det	149	25	1.2	2.7	6	0.1
137	encode_one_block	145	7	0	1.9	4.1	17.6	177	FORD2	353	7	10.8	1.8	0	0.1
138	jpeg_fdct_islow	135	27	0	2.7	0	14.5	178	filt	410	12	0.2	2.5	6.8	0.1
139	emit_bits	82	7	0	1.7	3.7	8.5	179	fft_pow	443	15	0	2.1	9	0.1