Survey on Combinatorial Register Allocation and Instruction Scheduling

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

Register allocation (mapping variables to processor registers or memory) and instruction scheduling (reordering instructions to increase instruction-level parallelism) are essential tasks for generating efficient assembly code in a compiler. In the past three decades, combinatorial optimization has emerged as an alternative to traditional, heuristic algorithms for these two tasks. Combinatorial optimization approaches can deliver optimal solutions according to a model, can precisely capture trade-offs between conflicting decisions, and are more flexible at the expense of increased compilation time.

This article provides an exhaustive literature review and a classification of combinatorial optimization approaches to register allocation and instruction scheduling, with a focus on the techniques that are most applied in this context: integer programming, constraint programming, partitioned Boolean quadratic programming, and enumeration. Researchers in compilers and combinatorial optimization can benefit from identifying developments, trends, and challenges in the area; compiler practitioners may discern opportunities and grasp the potential benefit of applying combinatorial optimization.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Retargetable compilers**; *Assembly languages*; • **Theory of computation** → **Constraint and logic programming**; **Mathematical optimization**; *Algorithm design techniques*;

Additional Key Words and Phrases: Combinatorial optimization, register allocation, instruction scheduling

ACM Reference format:
Roberto Castañeda Lozano and Christian Schulte. 2019. Survey on Combinatorial Register Allocation and Instruction Scheduling. *ACM Comput. Surv.* 52, 3, Article 62 (June 2019), 50 pages.
https://doi.org/10.1145/3200920

1 INTRODUCTION

Compiler back-ends take an intermediate representation (IR) of a program and generate assembly code for a particular processor. The main tasks in a back-end are instruction selection, register allocation, and instruction scheduling. Instruction selection implements abstract operations with

This article is a revised and extended version of a technical report [28].
This work has been partially funded by Ericsson AB and the Swedish Research Council (VR) under grant 621-2011-6229.
Authors' addresses: R. Castañeda Lozano, RISE SICS, Box 1263, Kista, 164 40, Sweden, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Electrum 229, Kista, 164 40, Sweden; email: roberto.castaneda@ri.se; C. Schulte, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Electrum 229, Kista, 164 40, Sweden, RISE SICS, Box 1263, Kista, 164 40, Sweden; email: cschulte@kth.se.

0360-0300/2019/06-ART62 $15.00
https://doi.org/10.1145/3200920

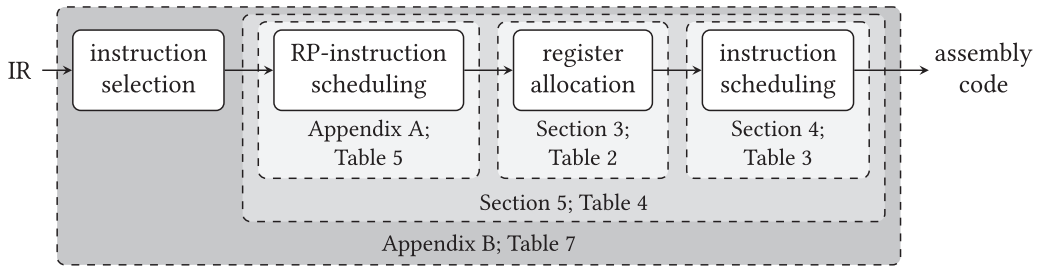


Fig. 1. Compiler back-end with section and table references.

processor instructions. Register allocation maps temporaries (program and compiler-generated variables in the IR) to processor registers or to memory. Instruction scheduling reorders instructions to improve the total latency or throughput. This survey is concerned with combinatorial approaches (explained in this section) for register allocation and instruction scheduling. Combinatorial instruction selection approaches are reviewed elsewhere [80].

Register allocation and instruction scheduling are of paramount importance to optimizing compilers [59, 78, 125]. In general, problems for these tasks are computationally complex (NP-hard) and interdependent: the solution to one of them affects the other [66]. Solving instruction scheduling first tends to increase the *register pressure* (number of temporaries that need to be stored simultaneously), which may degrade the result of register allocation. Conversely, solving register allocation first tends to increase the reuse of registers, which introduces additional dependencies between instructions and may degrade the result of instruction scheduling [68].

Heuristic approaches. Traditional back-ends solve each problem in isolation with custom heuristic algorithms, which take a sequence of greedy decisions based on local optimization criteria. This arrangement makes traditional back-ends fast but precludes solving the problems optimally and complicates exploiting irregular architectures. Classic heuristic algorithms are graph coloring [29] for register allocation and list scheduling [136] for instruction scheduling. A typical scheme to partially account for the interdependencies between instruction scheduling and register allocation in this setup is to solve a register pressure (*RP*)-aware version of instruction scheduling before register allocation [66], as shown in Figure 1. Heuristic algorithms that further approximate this integration have also been proposed [21, 25, 119, 130].

Combinatorial approaches. Numerous approaches that use combinatorial optimization techniques to overcome the limitations in traditional back-ends have been presented starting in the 1980s [106]. Combinatorial approaches can solve compiler back-end problems optimally according to a model at the expense of increased compilation time, and their declarative nature provides more flexibility. The accuracy with which a combinatorial approach models its problem is key, as the computed solutions are only optimal with respect to the model rather than the problem itself. Recent progress in optimization technology and improved understanding of the structure of back-end problems allow us today to solve optimal register allocation and instruction scheduling problems of practical size in the order of seconds, as this survey illustrates. Furthermore, combinatorial approaches can precisely capture the interdependencies between different back-end problems to generate even better solutions, although doing it efficiently remains a major computational challenge. Combinatorial approaches might never fully replace traditional approaches due to their high computation cost, however they can act as a complement rather than a replacement. Given that combinatorial approaches precisely capture interdependencies, they can be used to experiment with new ideas as well as evaluate and possibly improve existing heuristics used in traditional

approaches; for example, Ericsson uses UNISON (see Section 5.2 for a discussion) for that purpose, as can be seen from an entry on their research blog [160].

For consistency and ease of comparison, this survey focuses on combinatorial techniques that use a general-purpose modeling language. These include integer programming [126], constraint programming [140], and partitioned Boolean quadratic programming [142]. A uniform treatment of integer programming and constraint programming is offered by Hooker [82]. For completeness, the survey also includes the most prominent special-purpose enumeration techniques, which are often founded in methods such as dynamic programming [35] and branch-and-bound search [126].

Contributions. This article reviews and classifies combinatorial optimization approaches to register allocation and instruction scheduling. It is primarily addressed to researchers in compiler and combinatorial optimization who can benefit from identifying developments, trends, and challenges in the area but may also help compiler practitioners to discern opportunities and grasp the potential benefit of applying combinatorial optimization. To serve these goals, the survey contributes:

- an overview of combinatorial optimization techniques used for register allocation and instruction scheduling with a focus on the most relevant aspects of these problems (Section 2);
- an exhaustive literature review of combinatorial approaches for register allocation (Section 3), instruction scheduling (Section 4), and the integration of both problems (Section 5); and
- a classification of the reviewed approaches (Tables 2, 3, and 4) based on technique, scope, problem coverage, approximate scalability, and evaluation method.

In addition, Appendices A and B review and classify (Tables 5 and 7) register pressure-aware instruction scheduling and integrated approaches to the three compiler back-end tasks. The article complements available surveys of register allocation [91, 122, 128, 133, 134], instruction scheduling [2, 42, 68, 136, 139], and integrated code generation [97], whose focus tends to be on heuristic approaches.

2 COMBINATORIAL OPTIMIZATION

Combinatorial optimization is a collection of *complete* techniques to solve combinatorial problems. *Combinatorial* refers to the problems' nature that the value combinations in their solutions must satisfy properties that are mutually interdependent. Not all combinatorial optimization problems are NP-hard, even though general scheduling and register allocation problems are. Relaxations of these problems—for example by dropping the optimality requirement—might also be solvable in polynomial time.

Complete techniques automatically explore the full solution space and guarantee to eventually find the optimal solution to a combinatorial problem—or prove that there is no solution at all. For consistency and ease of comparison among different approaches, this survey focuses on those combinatorial optimization techniques that provide support for describing the problem at hand with a general-purpose modeling language. This category is composed of a wide range of techniques often presenting complementary strengths, as illustrated in this survey. Those that are most commonly applied to code generation are Integer Programming (IP), Constraint Programming (CP), and, to a lesser extent, Partitioned Boolean Quadratic Programming (PBQP). This section reviews the modeling and solving aspects of these techniques, as well as the common solving methods in special-purpose enumeration techniques.

Section 2.1 presents the modeling language provided by IP, CP, and PBQP. Section 2.2 describes the main solving methods of each combinatorial technique with a focus on methods applied by the reviewed approaches.

Table 1. Modeling Elements for Different Techniques

technique	variables	constraints	objective function
IP	$x_i \in \mathbb{Z}$	$\sum_{i=1}^{n} a_i x_i \leq b$	$\sum_{i=1}^{n} c_i x_i$
		($a_i, b, c_i \in \mathbb{Z}$ are constant coefficients)	
CP	$x_i \in \mathbb{D}$	any $r(x_1, x_2, \ldots, x_n)$	any $f(x_1, x_2, \ldots, x_n)$
		($\mathbb{D} \subset \mathbb{Z}$ is a finite subset of the integers)	
PBQP	$x_i \in \mathbb{D}$	none	$\sum_{i=1}^{n} c(x_i) + \sum_{i,j=1}^{n} C(x_i, x_j)$
		($\mathbb{D} \subset \mathbb{Z}$ is a finite int. subset; $c(x_i)$ is the cost of x_i; $C(x_i, x_j)$ is the cost of $x_i \wedge x_j$)	

2.1 Modeling

Combinatorial models consist—regardless of the particular optimization technique discussed in this survey—of *variables*, *constraints*, and an *objective function*. *Variables* capture decisions that are combined to form a solution to a problem. Variables can take values from different domains (for example, integers $\mathbb{Z}$ or subsets of integers such as Booleans $\{0, 1\}$). The variables in a model are denoted here as $x_1, x_2, \ldots, x_n$. *Constraints* are relations over the values for the variables that must hold for a solution to a problem. The set of constraints in a model defines all legal combinations of values for its variables. The types of constraints that can be used depend on each combinatorial optimization technique. The *objective function* is an expression on the model variables to be minimized by the solving method. We assume without loss of generality that the objective function is to be minimized. The term *model* in this survey refers to combinatorial models unless otherwise stated.

Integer Programming (IP). IP is a special case of Linear Programming (LP) [157] where the variables range over integer values, the constraints are linear inequalities (which can also express linear equalities), and the objective function is linear, as shown in Table 1. Most compiler applications use bounded variables (with known lower and upper bounds that are parametric with respect to the specific problem being solved) and variables that range over $\{0, 1\}$ (called 0-1 variables). IP models are often called *formulations* in the literature. For an overview, see for example the classic introduction by Nemhauser and Wolsey [126].

Constraint Programming (CP). CP models can be seen as a generalization of bounded IP models where the variables take values from a finite subset $\mathbb{D} \subset \mathbb{Z}$ of the integers (including 0-1 variables), and the constraints and the objective function are expressed by general relations. CP typically supports a rich set of constraints over $\mathbb{D}$ including arithmetic and logical constraints but also constraints to model more specific subproblems such as assignment, scheduling, graphs, and bin-packing. Often, these more specific constraints are referred to as *global constraints* that express recurrent substructures involving several variables. Global constraints are convenient for modeling, but more importantly, are key to solving, as these constraints have constraint-specific efficient and powerful implementations. The solution to a CP model is an assignment of values to the variables such that all constraints are satisfied. More information on CP can be found in a handbook edited by Rossi et al. [140].

Partitioned Boolean Quadratic Programming (PBQP). PBQP is a special case of the Quadratic Assignment Problem [105] that was specifically developed to solve compiler problems with constraints involving up to two variables at a time [47, 48, 142]. As such, it is not as widely spread as

other combinatorial optimization techniques such as IP and CP, but this section presents it at the same level for uniformity. As with CP, variables range over a finite subset $\mathbb{D} \subset \mathbb{Z}$ of the integers. However, PBQP models do not explicitly formulate constraints but define problems by a quadratic cost function. Each single variable assignment x_i is given a cost $c(x_i)$, and each pair of variable assignments $x_i \wedge x_j$ is given a cost $C(x_i, x_j)$. Single assignments and pairs of assignments can then be forbidden by setting their cost to a conceptually infinite value. The objective function is the combination of the cost of each single assignment and the cost of each pair of assignments, as shown in Table 1. PBQP is described by Scholz et al. [76, 142]; more background information can be found in Eckstein's doctoral dissertation [46].

2.2 Solving Methods

Integer Programming. The most common approach for IP solvers is to exploit *linear relaxations* and *branch-and-bound search.* State-of-the-art solvers, however, exploit numerous other methods [126]. A first step computes the optimal solution to a relaxed LP problem, where the variables can take any value from the set $\mathbb{R}$ of real numbers. LP relaxations can be derived directly from the IP models, as these only contain linear constraints and are computed efficiently. If all the variables in the solution to the LP problem are integers (they are said to be integral), then the optimal solution to the LP relaxation is also optimal for the original IP model. Otherwise, the basic approach is to use branch-and-bound search that decomposes the problem into alternative subproblems in which a non-integral variable is assigned different integer values and the process is repeated. Modern solvers use a number of improvements such as cutting-plane methods, in particular Gomory cuts, which add linear inequalities to remove non-integer parts of the search space [126]. LP relaxations provide lower bounds on the objective function that are used to prove optimality. Solutions found during solving provide upper bounds that are used to discard subproblems that cannot produce better solutions.

Constraint Programming. CP solvers typically proceed by interleaving *constraint propagation* and *branch-and-bound search.* Constraint propagation reduces the search space by discarding values for variables that cannot be part of any solution. Constraint propagation discards values for each constraint in the model iteratively until no more values can be discarded [22]. Global constraints play a key role in solving, as they are implemented by particularly efficient and effective propagation algorithms [156]. A key application area for CP is scheduling, in particular variants of cumulative scheduling problems where the tasks to be scheduled cannot exceed the capacity of a resource used by the tasks [12, 13]. These problems are captured by global scheduling constraints and implemented by efficient algorithms providing strong propagation. When no further propagation is possible, search tries several alternatives on which constraint propagation and search is repeated. The alternatives in search typically follow a heuristic to reduce the search space. As with IP solving, valid solutions found during solving are exploited by branch-and-bound search to reduce the search space [154].

Partitioned Boolean Quadratic Programming. Optimal PBQP solvers interleave *reduction* and *branch-and-bound search* [76]. Reduction transforms the original problem by iteratively applying a set of rules that eliminate one *reducible* variable at a time. Reducible variables are those related to at most two other variables by non-zero costs. If at the end of reduction the objective function becomes trivial (that is, only the costs of single assignments $c(x_i)$ remain), then a solution is obtained. Otherwise, branch-and-bound search derives a set of alternative PBQP subproblems on which the process is recursively repeated. The branch-and-bound method maintains lower and upper bounds on the objective function to prove optimality and discard subproblems as the search goes.

Properties and expressiveness. The solving methods for IP, CP, and PBQP all rely on branch-and-bound search. All techniques are in principle designed to be *complete*; that is, to find the best solution with respect to the model and objective function and to prove its optimality. However, all three approaches also support *anytime behavior*: the search finds solutions with increasing quality and can be interrupted at any time. The more time is allocated for solving, the better the found solution is.

The three techniques offer different trade-offs between the expressiveness of their respective modeling languages and their typical strength and weaknesses in solving.

IP profits from its regular and simple modeling language in its solving methods that exploit its regularity. For example, Gomory cuts generated during solving are linear inequalities themselves. IP is in general good at proving optimality due to its simple language and rich collection of global methods; in particular, relaxation and cutting-plane methods. However, the restricted expressiveness of the modeling language can sometimes result in large models, both in the number of variables as well as in the number of constraints. A typical example are scheduling problems that need to capture the order among tasks to be scheduled. Ordering requires disjunctions that are difficult to express concisely and can reduce the strength of the relaxation methods.

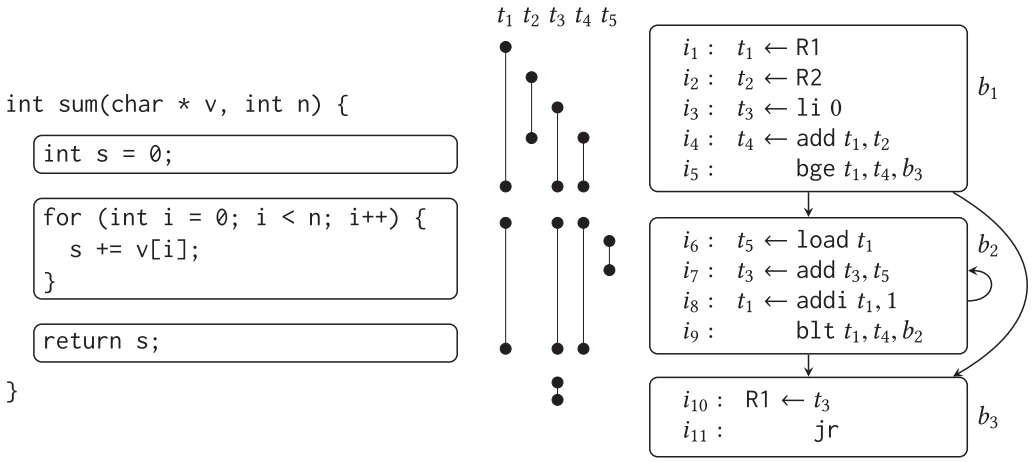
CP has somewhat complementary properties. CP is good at capturing structure in problems, typically by global constraints, due to its more expressive language. The individual structures are efficiently exploited for propagation algorithms specialized for a particular global constraint. However, CP has limited search capabilities compared to IP. For example, there is no natural equivalent to a Gomory cut, as the language is diverse and not regular and there is no general concept of relaxation. Recent approaches try to alleviate this restriction using methods from SAT (Boolean satisfiability) [127] techniques. CP is in general less effective at optimization. It might find a first solution quickly, but proving optimality can be challenging.

PBQP is the least-explored technique and has been mostly applied to problems in compilation. Its trade-offs are not obvious, as it does not offer any constraints but captures the problem by the objective function. In a sense, it offers a hybrid approach, as optimality for the objective function can be relaxed and hence the approach turns into a heuristic. Or it is used together with branch-and-bound search, which makes it complete while retaining anytime behavior.

Special-purpose enumeration. Special-purpose enumeration techniques define and explore a search tree where each node represents a partial solution to the problem. The focus of these techniques is usually in exploiting problem-specific properties to reduce the amount of nodes that need to be explored, rather than relying on a general-purpose framework such as IP, CP, or PBQP. Typical methods include merging equivalent partial solutions [98] in a similar manner to *dynamic programming* [35], detection of *dominated* decisions that are not essential in optimal solutions [135], branch-and-bound search [144], computation of lower bounds [108, 138], and feasibility checks similar to constraint propagation in CP [144]. Developing special-purpose enumeration techniques incurs a significant cost but provides high flexibility in implementing and combining different solving methods. For example, while CP typically explores search trees in a depth-first search fashion, merging equivalent partial solutions requires breadth-first search [98].

3 REGISTER ALLOCATION

Register allocation takes as input a function where instructions of a particular processor have been selected. Functions are usually represented by their control-flow graph (CFG). A basic block in the CFG is a straight-line sequence of *instructions* without branches from or into the middle of the sequence. Instructions use and define *temporaries*. Temporaries are storage locations holding values corresponding to program and compiler-generated variables in the IR.



(a) C source code (b) Live ranges and CFG

Fig. 2. Running example: sum function.

A *program point* is located between two consecutive instructions. A temporary t is *live* at a program point if t holds a value that might be used in the future. The *live range* of a temporary t is the set of program points where t is live. Two temporaries holding different values *interfere* if their live ranges overlap.

Figure 2(a) shows the C source code of a function returning the sum of the n elements of an array v. Figure 2(b) shows its corresponding CFG in the form taken as input to register allocation. In this form, temporaries t_1, t_2, and t_3 correspond directly to the C variables v, n, and s; t_4 corresponds to the end of the array (v + n); and t_5 holds the element loaded in each iteration. t_1, t_3, t_4, and t_5 interfere with each other and t_2 interferes with t_1 and t_3, as can be seen from the live ranges depicted to the left of the CFG. The example uses the following MIPS32-like instructions [149]: li (load immediate), add (add), addi (add immediate), bge (branch if greater or equal), load (load from memory), blt (branch if lower than), and jr (jump and return). The sum function is used as running example throughout this article.

Register allocation and assignment. Register allocation maps temporaries to either processor registers or memory. The former are usually preferred, as they have faster access times. *Multiple allocation* allows temporaries to be allocated to both memory and processor registers simultaneously (at the same program point), which can be advantageous in certain scenarios [34, Section 2.2]. *Register assignment* gives specific registers to register-allocated temporaries. The same register can be assigned to multiple, non-interfering temporaries to improve register utilization.

Spilling. In general, the availability of enough processor registers is not guaranteed and some temporaries must be *spilled* (that is, allocated to memory). Spilling a temporary t requires the insertion of store and load instructions to move t's value to and from memory. The simplest strategy (known as *spill-everywhere*) inserts store and load instructions at each definition and use of t. *Load-store optimization* allows t to be spilled at a finer granularity to reduce spill code overhead.

Coalescing. The input program may contain temporaries related by *copies* (operations that replicate the value of a temporary into another). Non-interfering copy-related temporaries can be *coalesced* (assigned to the same register) to discard the corresponding copies and thereby improve efficiency and code size. Likewise, copies of temporaries to or from registers (such as $t_1 \leftarrow$ R1

and R1 ← t_3 in Figure 2(b)) can be discarded by assigning the temporaries to the corresponding registers whenever possible.

Live-range splitting. Sometimes it is desirable to allocate a temporary t to different locations during different parts of its live range. This is achieved by *splitting* t into a temporary for each part of the live range that might be allocated to a different location.

Packing. Each temporary has a certain bit-width that is determined by its source data type (for example, `char` versus `int` in C). Many processors allow several temporaries of small widths to be assigned to different parts of the same register of larger width. This feature is known as *register aliasing*. For example, Intel's x86 [89] combines pairs of 8-bit registers (AH, AL) into 16-bit registers (AX). Packing non-interfering temporaries into the same register is key to improving register utilization.

Rematerialization. In processors with a limited number of registers, it can sometimes be beneficial to recompute (that is, *rematerialize*) a value to be reused rather than occupying a register until its later use or spilling the value.

Multiple register banks. Some processors include multiple register banks clustered around different types of functional units, which often leads to alternative temporary allocations. To handle these architectures effectively, register allocation needs to take into account the cost of allocating a temporary to different register banks and moving its value across them.

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

Evaluation methods. Combinatorial approaches to code generation tasks can be evaluated statically (based on a cost estimation by the objective function), dynamically (based on the actual cost from the execution of the generated code), or by a mixture of the two (based on a static cost model instantiated with execution measurements). For runtime objectives such as speed, the accuracy of static evaluations depends on how well they predict the behavior of the processor and benchmarks. For register allocation, dynamic evaluations are usually preferred, since they are most accurate and capture interactions with later tasks such as instruction scheduling. Mixed evaluations tend to be less accurate but can isolate the effect of register allocation from other tasks. Static evaluations require less implementation effort and are suitable for static objectives (such as code size minimization) or when an execution platform is not available.

Outline. Table 2 classifies combinatorial register allocation approaches with information about their optimization technique, scope, problem coverage, approximate scalability, and evaluation method.[1] *Problem coverage* refers to the subproblems that each approach solves in integration with combinatorial optimization. Approaches might exclude subproblems for scalability, modeling purposes, or because they do not apply to their processor model. The running text discusses the motivation behind each approach. Scalability in this classification is approximated by the size of the largest problem solved optimally, as reported by the original publications. Question marks are used when this figure could not be retrieved (no reevaluation has been performed in the scope of this survey). Improvements in combinatorial solving and increased computational power should be taken into account when comparing approaches over time.

[1]For simplicity, Table 2 classifies mixed evaluations as dynamic.

Table 2. Register Allocation Approaches

approach	TC	SC	SP	RA	CO	LO	RP	LS	RM	MB	MA	SZ	DE
ORA	IP	global	●	●	●	●	●	●	●	●	●	~2,000	●
Scholz and Eckstein [142]	PBQP	global	●	●	●	○	●	○	●	○	○	~200	●
PRA	IP	global	●	●	○	●	○	●	●	●	○	?	●
SARA	IP	global	●	●	○	●	○	●	○	○	○	?	●
Barik et al. [15]	IP	global	●	●	○	●	●	●	●	●	○	302	○
Naik and Palsberg [121]	IP	global	○	○	○	○	○	○	○	●	○	850	○
Falk et al. [56]	IP	global	●	●	○	●	○	●	○	●	●	~1,000	●
Appel and George [7]	IP	global	●	○	○	●	○	●	○	○	○	~2,000	●
Ebner et al. [45]	IP	global	●	○	○	●	○	●	○	○	○	?	●
Colombet et al. [34]	IP	global	●	○	○	●	○	●	●	○	●	?	●

Technique (TC), scope (SC), spilling (SP), register assignment (RA), coalescing (CO), load-store optimization (LO), register packing (RP), live-range splitting (LS), rematerialization (RM), multiple register banks (MB), multiple allocation (MA), size of largest problem solved optimally (SZ) in number of instructions, and whether a dynamic evaluation is available (DE).

Section 3.1 covers the first approaches that include register assignment as part of their combinatorial models, forming a baseline for all subsequent combinatorial register allocation approaches. Sections 3.2 and 3.3 cover the study of additional subproblems and alternative optimization objectives. Section 3.4 discusses approaches that decompose register allocation (including spilling) and register assignment (including coalescing) for scalability. Section 3.5 closes with a summary of developments and challenges in combinatorial register allocation.

3.1 Basic Approaches

Optimal Register Allocation. Goodwin and Wilken introduce the first widely recognized approach to combinatorial register allocation [67], almost three decades after some early work in the area [38, 112]. The approach, called *Optimal Register Allocation* (ORA), is based on an IP model that captures the full range of register allocation subproblems (see Table 2). Goodwin and Wilken's ORA demonstrated, for the first time, that combinatorial global register allocation is feasible—although slower than heuristic approaches.

The ORA allocator derives an IP model in several steps. First, a *temporary graph* (Goodwin and Wilken refer to temporaries as *symbolic registers*) is constructed for each temporary t and register r where the nodes are the program points $p_1, p_2, \ldots, p_n$ at which t is live and the arcs correspond to possible control transitions. Then, the program points are annotated with register allocation decisions that correspond to 0-1 variables in the IP model and linear constraints involving groups of decisions. Figure 3 shows the temporary graph corresponding to t_1 and R1 in the running example.

The model includes four main groups of variables to capture different subproblems, where each variable is associated to a specific program point p in the temporary graph: *register assignment variables* $\mathrm{def}(t, r, p)$, $\mathrm{use\text{-}cont}(t, r, p)$, and $\mathrm{use\text{-}end}(t, r, p)$ indicate whether temporary t is assigned to r at each definition and use of t (use-cont and use-end reflect whether the assignment is effective at the use point and in that case whether it continues or ends afterwards); *spilling variables* $\mathrm{store}(t, r, p)$, $\mathrm{cont}(t, r, p)$, and $\mathrm{load}(t, r, p)$ indicate whether temporary t (which is assigned to register r) is stored in memory, whether the assignment to r continues after a possible store, and whether t is loaded from memory to r; *coalescing variables* $\mathrm{elim}(t, t', r, p)$ indicate whether the copy from t to t' is eliminated by assigning t and t' to r; and *rematerialization variables* $\mathrm{remat}(t, r, p)$ indicate whether t is rematerialized into r. In the original notation, each variable is prefixed by x

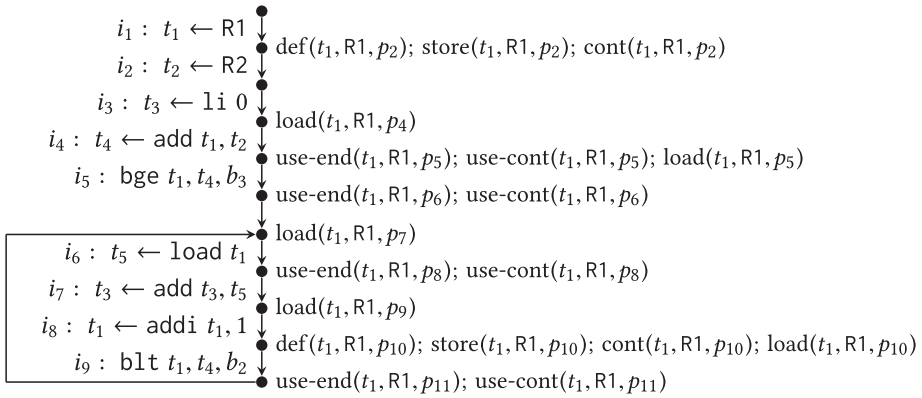


Fig. 3. Simplified ORA temporary graph for t_1 and R1.

and suffixed and superscripted by its corresponding register and temporary.[2] Figure 3 shows the variables for t_1 and R1 at different program points.

The model includes linear constraints to enforce that: at each program point, each register holds at most one temporary; each temporary t is assigned to a register at t's definition and uses; each temporary is assigned the same register where its live ranges are merged at the join points of the CFG; and an assignment of temporary t to a register that holds right before a use is conserved until the program point where t is used. For example, the temporary graph shown in Figure 3 induces the constraint use-cont(t_1, R1, p_5) + use-end(t_1, R1, p_5) = cont(t_1, R1, p_2) + load(t_1, R1, p_4) to enforce that the assignment of t_1 to R1 can only continue or end at program point p_5 (after i_4) if t_1 is actually assigned to R1 at that point. Other constraints to capture spilling, coalescing, and rematerialization are listed in the original article [67].

The objective function minimizes the total cost of decisions reflected in the spilling, coalescing, and rematerialization variables. In the running example, the store(t_1, R1, p) and load(t_1, R1, p) variables are associated with the estimated cost of spilling at each program point p where they are introduced (based on estimated execution frequency and type of spill instructions) while def(t_1, R1, p_2) is associated with the estimated benefit of discarding the copy i_1 by coalescing t_1 and R1.

Goodwin and Wilken use a commercial IP solver and compare the results against those of GCC's [62] register allocator for a Hewlett-Packard PA-RISC processor [92]. Their experiments reveal that, in practice, register allocation problems have a manageable average complexity, and functions of hundreds of instructions can be solved optimally in a time scale of minutes.

The results of Goodwin and Wilken encouraged further research based on the ORA approach. Kong and Wilken present a set of extensions to the original ORA model, including register packing and multiple register banks, to deal with irregularities in register architectures [104]. The extensions are complete enough to handle Intel's x86 [89] architecture, which presents a fairly irregular register file. Kong and Wilken estimate that their extended ORA approach reduces GCC's execution time overhead due to register allocation by 61% on average. The estimation is produced by a mixed static-dynamic evaluation that instantiates the model's objective function with the actual execution count of spill, coalescing, and rematerialization instructions. While this estimation is more accurate than a purely static one, a study of its relation to the actual execution time is not

[2]The original variable and constraint names in the reviewed publications are sometimes altered for clarity, consistency, and comparability. A note is made whenever this is the case.

available. Besides improving code quality, Kong and Wilken speed up the solving time of Goodwin and Wilken by two orders of magnitude. The reasons behind this speedup are a reduction of the search space due to both the availability of fewer registers and the introduction of irregularities, and the use of a faster machine with a newer version of the IP solver. The results illustrate an interesting aspect of combinatorial optimization: factors that complicate the design of heuristic approaches such as processor irregularities do not necessarily affect combinatorial approaches negatively—sometimes quite the opposite.

Fu and Wilken reduce the (still large) solving time gap between the ORA and heuristic approaches [61]. Their faster ORA approach identifies numerous conditions under which decisions are dominated (that is, provably suboptimal). For example, the fact that the ORA model assumes a constant cost for any placement of spill code within the same basic block makes certain spilling decisions dominated. The variables corresponding to such decisions are guaranteed to be zero in some optimal solution and can thus be discarded to reduce the model's complexity.

Fu and Wilken find that the solving process is sped up by roughly four orders of magnitude compared to the original ORA approach: two due to increased computational power and algorithmic improvements in the IP solver during the six-year gap between the publications, and two due to the removal of dominated variables and their corresponding constraints. According to their results, the improvements make it possible to solve 98.5% of the functions in the SPEC92 integer benchmarks [147] optimally with a time limit of 1,024s.

Scholz et al. Scholz and Eckstein propose an alternative combinatorial approach that models register allocation as a partitioned Boolean quadratic programming (PBQP) problem [142]. The simplicity with which register allocation can be reduced to a PBQP problem and the availability since 2008 of a production-quality implementation in the LLVM compiler [109] have made PBQP a popular technique for this purpose. However, the simplicity of this approach comes with limitations—the range of subproblems that are captured is narrower than that of the more general ORA approach (see Table 2). Although more subproblems could, in principle, be modeled with additional variables and costs, it remains an open question whether the resulting scalability would match that of IP-based approaches.

In contrast to the rather sophisticated ORA model, Scholz and Eckstein's model features a single class of variables $\mathrm{a}(t)$ giving the register to which temporary t is assigned. The decisions to spill a temporary t to memory or to rematerialize it [75, Chapter 4] are captured by including special *spilling* and *rematerialization registers* $\mathtt{sp}, \mathtt{rm}$ to the domain of its variable $\mathrm{a}(t)$. In the original notation, each variable $\mathrm{a}(t)$ is defined as a collection of alternative Boolean variables $\{\mathrm{x}(t, \mathtt{R0}), \mathrm{x}(t, \mathtt{R1}), \ldots, \mathrm{x}(t, \mathtt{sp}), \mathrm{x}(t, \mathtt{rm})\}$ where each Boolean variable captures exactly one value of $\mathrm{a}(t)$ and $\mathrm{a}(t)$ is referred to as a vector $\vec{x}(t)$. As is characteristic of PBQP models (see Table 1), constraints are defined by giving conceptually infinite costs to forbidden single assignments $c(\mathrm{a}(t))$ and pairs of assignments $C(\mathrm{a}(t), \mathrm{a}(t'))$. Individual costs $c(\mathrm{a}(t))$ are used to forbid the assignment of temporary t to registers that do not belong to its supported register classes, account for the overhead of spilling or rematerializing t, and account for the benefit of coalescing t with preassigned registers. Costs of pairs of assignments $C(\mathrm{a}(t), \mathrm{a}(t'))$ are used to forbid assignments of interfering temporaries to the same (or aliased) registers, and account for the benefit of coalescing t and t'. The objective function minimizes the cost given to single assignments and pairs of assignments, thus avoiding solutions forbidden by conceptually infinite costs.

Figure 4 shows the assignment costs for the running example from Figure 2, where c is the estimated benefit of discarding a copy by coalescing and s is the estimated cost of spilling a temporary (uniform benefits and costs are assumed for simplicity). None of the temporaries can be rematerialized, since their values cannot be recomputed from available values [29], hence the cost

	R1	R2	···	R31	sp	rm
$a(t_1)$ $a(t_3)$	$-c$	0	···	0	s	∞
$a(t_2)$	0	$-c$	···	0	s	∞
$a(t_4)$ $a(t_5)$	0	0	···	0	s	∞

(a) Costs of individual assignments

		$a(t_1)$					
		R1	R2	···	R31	sp	rm
$a(t_2)$	R1	∞	0	···	0	0	0
	R2	0	∞	···	0	0	0
	···	···	···	···	···	···	···	···
	R31	0	0	···	∞	0	0
	sp	0	0	···	0	0	0
	rm	0	0	···	0	0	0

(b) Costs of assignments for t_1 and t_2

Fig. 4. Assignment costs in Scholz and Eckstein's model for the running example.

of assigning them to rm is infinite. Since t_1 and t_2 interfere, assignments to the same register incur an infinite cost. This is the case for all pairs of temporaries in the example except (t_2, t_4) and (t_2, t_5), which yield null matrices, as they do not interfere.

Scholz and Eckstein propose both a heuristic and an optimal PBQP solver. When no reduction rule applies, the former applies applies greedy elimination rules while the latter resorts to exhaustive enumeration (as opposed to branch-and-bound search). As this survey is concerned with combinatorial approaches, only results related to the optimal PBQP solver are discussed. Scholz and Eckstein experiment with five signal processing benchmarks on Infineon's Carmel 20xx [87] Digital Signal Processor (DSP) to demonstrate the ease with which its special register allocation constraints (which dictate register combinations allowed for certain pairs of temporaries) are modeled in PBQP. Their dynamic evaluation shows that the optimal solver can deliver up to 13.6% faster programs than graph-coloring-based heuristics [146]. A complementary static evaluation of the optimal PBQP solver by Hirnschrott et al. supports the conclusions of Scholz and Eckstein for different versions of an ideal DSP with different numbers of registers and instruction operands [79].

Hames and Scholz extend the PBQP solver, based originally in exhaustive enumeration, with a branch-and-bound search mechanism to reduce the amount of search needed to find optimal solutions [76]. Hames and Scholz's static evaluation on Intel's x86 [89] shows that their branch-and-bound PBQP solver solves 97.4% of the SPEC2000 [147] functions over 24 hours, yielding a slight estimated spill cost reduction of 2% over the same heuristic approach as in the original experiments. This suggests the improvement potential of Scholz et al.'s approach over heuristics is limited for general-purpose processors and larger for more constrained processors such as DSPs.

Progressive Register Allocation. Koes and Goldstein introduce a *progressive* register allocation (PRA) approach [101, 102]. An ideal progressive solver should deliver reasonable solutions quickly, find improved solutions if more time is allowed, and find an optimal solution if enough time is available. Although both the ORA and Scholz et al.'s approaches can also potentially behave progressively, in 2005 none of them was able to meet the three conditions.

Koes and Goldstein propose modeling register allocation as a multi-commodity network flow (MCNF) problem [1], which can be seen as a special case of an IP problem. The MCNF problem consists of finding a flow of multiple commodities through a network such that the cost of flowing through all arcs is minimized and the flow capacity of each arc is not exceeded. The reduction of register allocation to MCNF is intuitive: each commodity corresponds to a temporary that flows through storage locations (registers and memory) at each program point, and the network's structure forces interfering temporaries to flow through different registers. This model can express detailed allocations and accurately take into account their cost. Furthermore, the reduction to MCNF enables exploiting well-understood techniques to solve network problems progressively. However,

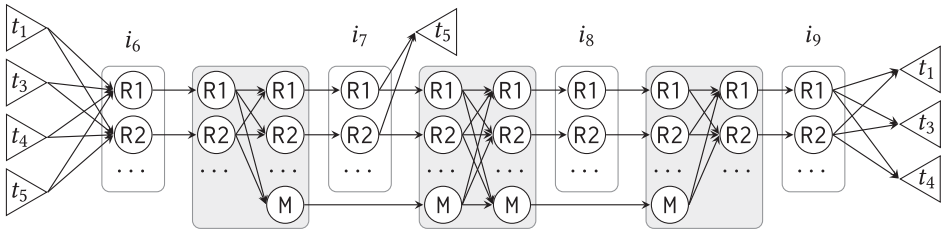


Fig. 5. Simplified multi-commodity network flow for basic block b_2 in the PRA model.

the flow abstraction cannot cope with coalescing, register packing, or multiple allocations of the same temporary, which makes the PRA model less general than that of the ORA approach.

In the PRA approach, each commodity in the network flow corresponds to a temporary. For each program point and storage location, a node is added to the network. The flow of a temporary t through the network determines how t is allocated. As with Scholz et al.'s model, a single class of variables is defined: $\mathrm{a}(t, i, j)$ ($x^t_{i,j}$ in the original notation) indicates whether the temporary t flows through the arc (i, j) where i, j represent storage locations at a program point. Compared to Scholz et al.'s model, the additional program point dimension makes it possible to express more detailed register allocations capturing live-range splitting. The model includes linear constraints to enforce that: only one temporary flows through each register node (called *bundle constraints* by Koes and Goldstein), and flow is conserved through the nodes (the same amount of flow that enters a node exits it). The objective function minimizes the arc traversal cost for all flows. The cost of an arc $\mathrm{c}(i, j)$ reflects the cost of moving a temporary from the source location i to the destination location j: if i and j correspond to the same locations, then no cost is incurred; if i and j correspond to different registers, then $\mathrm{c}(i, j)$ is the cost of a register-to-register move instruction; and if one of i and j corresponds to memory and the other to a register, then $\mathrm{c}(i, j)$ is the cost of a memory access instruction.

Figure 5 shows the MCNF corresponding to basic block b_2 in the running example from Figure 2. Temporary source and sink nodes are represented by triangles, while storage locations (registers R1, R2, ..., and memory M) are represented by circles. Each rectangle contains storage locations corresponding to either instructions or program points. The latter (colored in gray) allow temporaries to flow across different storage locations between the execution of two instructions. The MCNF is constructed to force temporaries used by an instruction to flow through the storage locations supported by the instruction. Unused temporaries can bypass instruction storage locations by following additional arcs between gray rectangles (not depicted in Figure 5 for clarity). All arcs have capacity one except arcs between memory nodes. Such arcs are uncapacitated, allowing any number of temporaries to be simultaneously spilled.

Although the PRA model is a valid IP model and can thus be solved by a regular IP solver, Koes and Goldstein propose a dedicated solving scheme to attain a more progressive behavior. The scheme is based on a Lagrangian relaxation, a general IP technique that, similarly to PBQP models, replaces hard constraints by terms in the objective function that penalize their violation. Relaxing the bundle constraints allows finding solutions heuristically through shortest-path computations on the network. The Lagrangian relaxation is used to guide the heuristics towards improving solutions in an iterative process. This solving scheme is not complete, but in practice it can often prove optimality using bounds derived from the Lagrangian relaxation.

Koes and Goldstein compare their progressive solver with GCC's [62] graph-coloring register allocator and a commercial IP solver. Their experiments with functions from different benchmarks show that the PRA approach is indeed more progressive than standard IP: it delivers first

solutions in a fraction of the time taken by the IP solver and solves 83.5% of the functions optimally after 1,000 iterations. Koes and Goldstein report that their optimal solutions yield an average code size reduction of 6.8% compared to GCC's heuristic approach.

3.2 Model Extensions

Further research on combinatorial register allocation has addressed extensions of the baseline established by the basic approaches to cope with different processor features.

Stack allocation is the problem of assigning specific stack locations to spilled temporaries. This problem is typically solved after register allocation; however, some processors provide features that can be best exploited if both problems are solved in integration. *SARA* [124] is an IP approach that integrates register allocation with stack allocation to exploit the double-store and double-load instructions available in some ARM [8] processors. Such instructions can be used to spill pairs of temporaries but impose additional constraints on the register assignment and stack allocation of the spilled pairs, hence motivating the integrated SARA approach. SARA's IP model is composed of a basic register allocation submodel with load-store optimization and live-range splitting (see Table 2) and a stack allocation submodel. The latter includes *location variables* $\mathrm{f}(t, l)$ indicating whether temporary t is allocated to stack location l, and explicit *load and store pair variables* load-pair(i, t_1, t_2) and store-pair(i, t_1, t_2) indicating whether temporaries t_1, t_2 form a spill pair. Linear constraints enforce that spilled temporaries are given a single location, locations are not reused by multiple spilled temporaries, and spill pairs satisfy the register assignment and stack allocation conditions. The objective function minimizes estimated spill cost. Nandivada and Palsberg's experiments for Intel's StrongARM processor on 179 functions from different benchmarks show that the integrated approach of SARA indeed generates faster code (4.1%) than solving each problem optimally but in isolation.

Bit-width-aware register allocation extends register allocation to handle processors that support referencing bit ranges within registers, which is seen as a promising way to reduce register pressure in multimedia and network processing applications [123, 150]. Handling such processors can be seen as a generalization of register packing where register parts can be accessed with the finest granularity and the bit-width of temporaries varies through the program. The only combinatorial approach to bit-width-aware register allocation is due to Barik et al. [15]. Their key contribution is an IP register allocation model that allows multiple temporaries to be assigned to the same register r simultaneously as long as the bit capacity of r is not exceeded. This is supported by generalizing the common constraints that ensure that each register is not assigned more than one temporary simultaneously into constraints that ensure that the sum of the bit-width of the temporaries assigned to each register r simultaneously does not exceed the capacity of r. The model does not capture the placement of temporaries within registers, and therefore it disregards the cost of *defragmenting* registers with register-to-register move instructions. Barik et al. perform a static evaluation with benchmarks from the MediaBench [110] and Bitwise [148] suites on an ideal processor with bitwise register addressing. Their results show that, for such processors and applications, extending combinatorial register allocation with bit-width awareness can reduce the amount of spilling notably at the expense of solving efficiency.

3.3 Alternative Optimization Objectives

While most combinatorial register allocation approaches are concerned with speeding up the average execution time of the generated code, certain domains such as embedded systems show a high interest in alternative optimization objectives such as minimizing code size, energy, or worst-case execution time.

Naik and Palsberg introduce an IP approach [121] to minimize code size for Zilog's Z86E30 [170] processor. This processor lacks stack memory and provides instead 16 register banks of 16 registers each, which necessitates whole-program register allocation without spilling. Unlike other combinatorial register allocation approaches, Naik and Palsberg assume that temporaries are always live and thus get dedicated registers. This assumption can increase register pressure significantly for programs containing many short-lived temporaries, but this increase might be acceptable for the Z86E30 processor due to its large amount of registers and the expected modest size of its targeted applications. Both the lack of memory and the full-liveness assumption reduce the amount of subproblems that need to be modeled significantly, as seen in Table 2. The Z86E30 instructions can address a register by either specifying its absolute address or an offset relative to a special register pointer. The latter mode is encoded with one byte less, creating an opportunity to improve code size during register allocation. For this purpose, Naik and Palsberg propose an IP model that integrates register bank assignment with management of the register pointer at different program points. The model includes *register bank assignment variables* $\mathrm{r}(t, b)$ indicating whether temporary t is assigned to register bank b, *current register pointer variables* $\text{rp-val}(p, b)$ indicating whether the register pointer is set to register bank b at program point p, and other variables to reflect the size of each instruction and the updates that are applied to the register pointer at different program points. Linear constraints enforce that each temporary is assigned to one register bank, the capacity of the register banks is not exceeded, and the register pointer is updated accordingly to its intended register bank at each program point. The objective function minimizes the total code size, including that of the additional instructions needed to update the register pointer. Naik and Palsberg's experiments show that their approach can match the size of hand-optimized code for two typical control applications.

Real-time applications are usually associated with timing requirements on their execution. In that context, worst-case execution time (WCET) minimization is an attractive optimization objective, since reducing the WCET of a real-time application allows its computational cost to be reduced without compromising the timing guarantees. Falk et al. present an IP approach for WCET minimization [56] in the broader context of a WCET-aware compiler. The approach extends the basic ORA model with an alternative objective function that minimizes the execution time of the longest execution path through the function. This is formulated by introducing *WCET variables* $\mathrm{w}(b)$ and *cost variables* $\mathrm{c}(b)$ corresponding to the execution time of each basic block b and of the longest execution path starting at b, and letting the objective function minimize $\mathrm{w}(b_e)$ where e is the entry basic block of the function. The WCET and cost variables are related with linear constraints over all paths of the function's CFG excluding arcs that form loops (called *back edges*). For example, the CFG from Figure 2(b) yields the constraints $\mathrm{w}(b_3) = \mathrm{c}(b_3)$; $\mathrm{w}(b_2) = \mathrm{w}(b_3) + n \times \mathrm{c}(b_2)$; and $\mathrm{w}(b_1) = \max(\mathrm{w}(b_2), \mathrm{w}(b_3)) + \mathrm{c}(b_1)$, where max is, in practice, linearized with two inequalities and n is the given maximum iteration count of b_2. The cost $\mathrm{c}(b)$ of a basic block b corresponds to the cycles it takes to perform all spills in b. The cycles of a spill are modeled accurately by exploiting the simplicity of the memory hierarchy of the target processor (Infineon TriCore [88]) and detailed pipeline knowledge. Falk et al.'s experiments on 55 embedded and real-time benchmarks with high register pressure show that their approach reduces the WCET by 14% compared to a heuristic WCET-aware register allocator and, as a side effect, speeds up code for the average case by 6.6%. Even though no problem sizes are reported, the authors find the solving time acceptable in the context of WCET-aware compilation.

3.4 Decomposed Approaches

Register allocation includes a vast amount of subproblems, yielding combinatorial problems of high average complexity, as seen through this section. In an effort to improve scalability, a line

of research has focused on decomposing combinatorial register allocation into groups of subproblems and solving each group optimality. The decomposition that has received most attention is solving spilling first (including strongly interdependent subproblems such as load-store optimization, live-range splitting, and rematerialization) followed by register assignment and coalescing. The key assumption behind this decomposition scheme is that the impact of spilling (in store and load instructions) is significantly higher than the impact of coalescing (in register-to-register move instructions), hence spilling is performed first. The decomposition improves scalability, because the size of the spilling model becomes independent of the number of registers, and it still delivers near-optimal solutions for processors where the assumption holds. For example, in an empirical investigation on the x86 and ARM processors, Koes and Goldstein confirm that solving spilling optimally and the remaining subproblems heuristically "has no discernible impact on performance" [103] compared to a non-decomposed, optimal approach. However, the solution quality can degrade for processors or objectives such as code size minimization where the impact of spilling does not necessarily dominate the cost [103]. Also, the decomposition precludes a full integration with instruction scheduling in a single combinatorial model, as both spilling and the remaining subproblems (register assignment and coalescing) have strong interdependencies with instruction scheduling. For decomposed approaches, Table 2 summarizes the first problem (spilling and associated subproblems) where by construction the register assignment, coalescing, and register packing subproblems do not apply.

Appel and George are the first to propose a decomposed scheme based on IP [7]. Their scheme solves spilling, load-store optimization, and live-range splitting optimally first and then solves register assignment and coalescing either optimally or with heuristic algorithms. The IP model for optimal spilling includes the variables $\mathrm{s}(t,p)$, $\mathrm{l}(t,p)$ $\mathrm{r}(t,p)$, $\mathrm{m}(t,p)$ to indicate whether each temporary t at each program point p is either stored to memory (s), loaded into a register (l), or kept in a register (r) or memory (m). The main model constraints ensure that the processor registers are not overused at any program point. The model also captures specific constraints of Intel's x86 to demonstrate the flexibility of the approach. The objective function minimizes the total cost of the spill code and the x86 instruction versions required by the register allocation. Appel and George's model is similar but not a strict subset of the ORA model, as the latter limits live-range splitting to a set of predetermined program points for scalability. The same authors also propose a simple IP model for solving register assignment and coalescing optimally; however, they report that the problems cannot be solved in reasonable time and resort to a heuristic approach in the experiments. Appel and George's experiments demonstrate that their approach indeed scales better than the initial ORA solver (a comparison with the improved ORA model is not available) and improves the speed of the code generated by a heuristic register allocator by 9.5%. To encourage further research on optimal register assignment and coalescing, the *Optimal Coalescing Challenge* [6] is proposed. A few years later, Grund and Hack present an IP approach that solves 90.7% of the challenge's coalescing problems (some of them with thousands of temporaries) optimally [73]. The approach reduces the input problem by preassigning temporaries to registers whenever it is safe and exploits the structure of the interference graph to derive additional constraints that speed up solving. Grund and Hack's approach is not included in Table 2, since it does not address the core register allocation problem.

Ebner et al. recognize that the problem tackled by Appel and George (spilling including live-range splitting and load-store optimization) can be modeled as a minimum cut problem with capacity constraints [45]. Such problems are well researched in the IP literature, and solvers can typically handle large instances efficiently [126]. Ebner et al. define a network where the nodes correspond to temporaries at particular program points and the arcs correspond to possible control transitions. A solution corresponds to a cut in the network where one partition is allocated to

memory and the other one to registers. The cost of a solution is the total cost of the arcs crossed by the cut, where the cost of an arc is the spill cost at its corresponding program point. The capacity constraints ensure that the processor registers are not overused at any program point. Ebner et al. solve the problem both with a dedicated solver based on a Lagrangian relaxation (where the capacity constraints are relaxed) and a commercial IP solver. Interestingly, their experiments show that the IP solver delivers optimal solutions in less solving time than all but the simplest configuration of the dedicated solver.

Finally, Colombet et al. introduce an alternative IP approach [34] that additionally captures rematerialization and multiple allocation and can handle programs in Static Single Assignment (SSA) form [37]. SSA is a program form that defines temporaries only once and explicates control-dependent definitions at join points of the CFG with special ϕ-instructions. SSA has proven itself useful for register allocation, as it enables register assignment in isolation to be solved optimally in polynomial time [74]. The basic IP model of Colombet et al. resembles that of Appel and George's but has the key difference that the variables corresponding to $\mathrm{r}(t,p)$ and $\mathrm{m}(t,p)$ are not mutually exclusive, allowing each temporary t to be allocated to both memory and a register at the same program point p. The basic model is extended with variables and constraints to handle rematerialization and some specifics of the SSA form. Colombet et al. compare experimentally their approach with that of Appel and George and the corresponding subset of the PRA model using the EEMBC [131] and SPEC2000 benchmarks for ST231 [57], a Very Long Instruction Word (VLIW) processor. The results estimate statically that the introduced approach yields significantly better results than Appel and George's (around 40%) and slightly better results (around 5%) than the adapted PRA approach. Around half of the estimated improvement over Appel and George's approach is due to supporting rematerialization, while the other half is mostly due to avoiding spurious store instructions by allocating temporaries to memory and registers simultaneously (the adapted PRA approach is not as penalized, since it assigns spurious store instructions a cost of zero). An interesting finding is that these substantial estimated improvements only correspond to modest runtime improvements, due to spill cost model inaccuracies and interactions with later compiler stages. Colombet et al. identify two stages that alter the cost estimated by their objective function significantly: *immediate folding* and instruction scheduling. The former removes immediate loads whenever the immediate can be directly encoded as an operand in its user instruction (which is common for rematerialization), while the latter tends to hide the cost of spill code, in particular for a VLIW processor such as ST231 where up to four instructions can be scheduled simultaneously. While the effect of immediate folding can be directly captured in a register allocation model, Colombet et al. leave it as an open question whether the effect of scheduling could be modeled without resorting to an integrated approach, as in Section 5.

3.5 Discussion

This section has reviewed combinatorial approaches to register allocation proposed within the past 20 years. Since the first proposed approach (ORA), combinatorial register allocation has been able to handle a wide range of subproblems for functions with up to a few thousands of instructions and to demonstrate actual code quality improvements on dynamic evaluations. Subsequent extensions for different processor features and optimization objectives illustrate the flexibility of the combinatorial approach, for which IP remains essentially unchallenged as the technique of choice. Further scalability with virtually no performance degradation can be attained by decomposing register allocation and focusing in solving spilling and its closest subproblems (load-store optimization, live-range splitting, ...) optimally, as pioneered by Appel and George's approach. These benefits come at the expense of flexibility: As observed by Koes and Goldstein, the approach is less

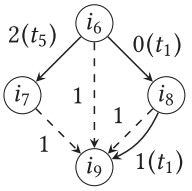


Fig. 6. DG of b_2.

effective when the remaining subproblems have a high impact on the quality of the solution, such as in code size optimization.

Despite its relative scalability and wide applicability, combinatorial register allocation is rarely applied in general-purpose production compilers. The single-digit average speedup demonstrated by most reviewed register allocation approaches is most likely not compelling enough for compilers aiming at striking a balance with compilation time. This raises two challenges for combinatorial register allocation: improving the quality of its generated code and reducing solving time. The first challenge calls for spill cost models that faithfully capture the effect of complex memory hierarchies (common in modern processors). Falk et al.'s approach takes a step in this direction. For VLIW processors, an open question is whether their effect can be captured by a combinatorial register allocation model without resorting to an integrated approach, as in Section 5. The second challenge could be addressed by multiple lines of research. The study of the structure of IP models for register allocation could result in significant scalability gains like in instruction scheduling (see Section 4). Alternative techniques such as CP have proven successful for other compiler problems but remain unexplored for register allocation. Hybrid combinatorial approaches such as PBQP that can resort to heuristics for large problems could allow compilers to benefit from combinatorial optimization and scale up with lower maintenance costs. Finally, combinatorial register allocation has potential to contribute to the areas of compiler validation, security, and energy efficiency due to its flexible yet formal nature.

4 INSTRUCTION SCHEDULING

Instruction scheduling maps program instructions to basic blocks and issue cycles within the blocks. A valid instruction schedule must satisfy the dependencies among instructions and cannot exceed the capacity of processor resources such as data buses and functional units. Typically, instruction scheduling aims at minimizing the *makespan* of the computed schedule—the number of cycles it takes to execute all its instructions. All approaches presented in this section have makespan minimization as their objective function unless otherwise stated.

Dependencies. Data and control flow cause dependencies among instructions. The dependencies in a basic block form a *dependency graph* (DG) where nodes represent instructions and an arc (i, j) indicates that instruction j depends on instruction i. Each arc (i, j) in a DG is labeled with the latency $\mathrm{l}(i, j)$ between i and j. The latency dictates the minimum amount of cycles that must elapse between the issue of the two instructions and is usually (but not necessarily) positive. For modeling convenience, DGs often have an *entry* (*exit*) instruction that precedes (succeeds) all other instructions.

Figure 6 shows the DG of the basic block b_2 in the running example from Figure 2, where i_6 is the entry instruction and i_9 is the exit instruction. Data dependencies are represented with solid arcs labeled with their latency (and corresponding temporary, for clarity). All instructions are assumed to have unit latency, except i_6 (`load`), which is assumed to have a latency of two cycles. (i_6, i_8) is called an *anti-dependency*: i_6 uses t_1, which is then redefined by i_8. Due to the structure of

instruction pipelines, instructions related by anti-dependencies can typically run simultaneously in multiple-issue processors (see the *Instruction bundling* paragraph in this section). Hence, such dependencies are often labeled with a latency of zero cycles. Branch dependencies maintain the original control flow (for example, forcing i_7 to be scheduled before i_9) and are represented with dashed arcs. Branch dependencies that are redundant from a latency perspective such as (i_6, i_9) are kept in the DG for uniformity.

Resources. Instructions share limited processor resources such as functional units and data buses. The organization of hardware resources varies widely among different processors and affects the complexity of instruction scheduling profoundly. Resources are classified as *single-* or *multi-capacity*, depending on the number of instructions that can access them at the same time. Instructions can use either one resource each (*single usage*) or multiple resources (*multiple usage*). Resource usage can be *uni-* or *two-dimensional,* depending on whether its duration is one or longer than one. Finally, resource usage is either *synchronous* if usage by an instruction i starts at the issue cycle of i or *asynchronous* otherwise. Asynchronous resource usage typically arises from irregularities in processor pipelines, where two instructions might contend for the same resource (for example, a pipeline stage) at a cycle different than their issues.

Instruction bundling. Classic RISC processors are *single-issue*: they can issue only one instruction at each clock cycle. Modern processors are usually *multiple-issue*: they can issue several instructions every clock cycle. To exploit this capability, in-order multiple-issue processors such as VLIW processors require the compiler to perform *instruction bundling*, combining instructions into bundles that are compatible with the issue restrictions of the processor. Such restrictions are often caused by constraints imposed by processor resources.

Scope. Unlike register allocation, which is typically approached globally, the instruction scheduling literature covers different problem scope levels, where each level can potentially deliver better code at the expense of increased complexity: *local instruction scheduling* schedules instructions within basic blocks in isolation, under the assumption that each instruction is already placed in a certain basic block; *regional instruction scheduling* considers collections of basic blocks with a certain CFG structure; and *global instruction scheduling* considers entire functions. Besides assigning each instruction i to an issue cycle within its block, regional and global instruction scheduling place i into a basic block.

Evaluation methods. In general, dynamic evaluations provide the highest accuracy for combinatorial instruction scheduling approaches that optimize runtime objectives such as speed. Static evaluations suit approaches that can capture the behavior of the program and processor statically (such as local approaches on in-order processors with constant instruction latencies).

Outline. Table 3 classifies combinatorial instruction scheduling approaches with information about their optimization technique, scope, problem coverage, approximate scalability, and evaluation method. The discussion is structured by scope: Sections 4.1, 4.2, and 4.3 cover local, regional, and global combinatorial instruction scheduling. Section 4.4 closes with a summary of developments and challenges in combinatorial instruction scheduling. Appendix A complements this section with a review of register-pressure-aware instruction scheduling approaches.

4.1 Local Instruction Scheduling

Local instruction scheduling is closely related to the resource-constrained project scheduling problem (RCPSP) [9], which aims at finding the minimal-makespan schedule of a set of precedence-related activities with resource demands such that the precedence and resource-capacity

Table 3. Instruction Scheduling Approaches

approach	TC	SC	BD	MU	2D	AS	SZ	DE
Arya [10]	IP	local	○	●	●	●	36	○
Ertl and Krall [55]	CP	local	○	●	●	●	24	●
Chou and Chung [32]	EN	local	●	○	○	○	20	○
Leupers and Marwedel [111]	IP	local	●	●	○	○	45	○
Wilken et al. [161]	IP	local	○	○	○	○	1,000	○
van Beek and Wilken [155]	CP	local	○	○	○	○	1,000	○
Malik et al. [115]	CP	local	●	●	○	○	2,600	○
Shobaki and Wilken [144]	EN	superblock	●	○	○	○	<1,236	○
Malik et al. [115]	CP	superblock	●	●	●	●	2,600	○
Beg and van Beek [18]	CP	superblock	●	●	●	●	100	○
Shobaki et al. [145]	EN	trace	●	○	○	○	<424	○
Govindarajan et al. [69]	IP	sw. pipelining	●	●	●	○	?	○
Altman et al. [4]	IP	sw. pipelining	●	●	●	●	~20	○
Winkel [166]	IP	global	●	●	●	○	600	●

Technique (TC), scope (SC where EN stands for *enumeration*), bundling (BD), multiple usage (MU), two-dimensional usage (2D), asynchronous usage (AS), size of largest problem solved optimally (SZ) in number of instructions, and whether a dynamic evaluation is available (DE).

constraints are satisfied. RCPSP solving is a well-researched area, and basic IP approaches were proposed as early as in the 1960s [24, 116, 132, 159]. Local instruction scheduling can be seen as an application of the general RCPSP that is concerned with specific code generation aspects such as complex processor resources and non-uniform instruction latencies [42].

Early approaches. An early combinatorial approach to local instruction scheduling was introduced by Arya [10]. This approach uses IP to compute optimal schedules for single-issue vector processors such as the early Cray-1 [141]. Arya's model can be seen as an application of Manne's RCPSP model [116] with general (that is, non 0-1) integer variables $\mathrm{s}(i)$ (T_i in the original notation) representing the issue cycle of each instruction i. This model allows dependency constraints to be formulated with simple inequalities on scheduling variables. For example, the dependency (i_6, i_7) in Figure 6 yields $\mathrm{s}(i_7) \geq \mathrm{s}(i_6) + 2$. However, the use of general integer variables precludes multi-capacity resources and requires auxiliary 0-1 variables $\mathrm{p}(i, j)$ to indicate whether instruction i precedes instruction j or vice versa. A constraint is added for each pair of instructions i, j to relate the auxiliary and the scheduling variables. For example, the corresponding constraint for i_7 and i_8 is $1 \leq M \times \mathrm{p}(i_7, i_8) + \mathrm{s}(i_7) - \mathrm{s}(i_8) \leq M - 1$, where M is a large positive constant. Each value of $\mathrm{p}(i_7, i_8)$ activates one of the sides of the compound inequality and enforces a precedence among i_7 and i_8 in one or the opposite direction. Arya shows on three basic blocks of up to 36 instructions that the approach is feasible and improves the makespan of hand-optimized code significantly.

Ertl and Krall introduced the first local instruction scheduling approach based on constraint programming [55]. In particular, the approach applies *constraint logic programming* [90], which embeds constraint propagation into logic programming. Their model targets single-issue, pipelined RISC processors, where the resource consumption of an instruction must be considered at each stage of the pipeline. The model includes, for each instruction i and pipeline stage k, a finite integer domain variable $\mathrm{s}(i, k)$ representing the cycle in which i resides in k (the original notation uses a set of variables $\{D_i, W_i, \dots\}$ for each instruction i, where each variable corresponds to a pipeline stage such as *decode* and *writeback*). Dependencies among instructions are modeled as in Arya's approach. Equality constraints link the different stage scheduling variables of an

instruction. An *all-different* global constraint [156] ensures that only one instruction resides in each pipeline stage k at a time by forcing all variables $\{s(i_1, k), s(i_2, k), \ldots, s(i_n, k)\}$ to take different values. The use of *all-different* yields a compact model, as it removes the need for a quadratic number of auxiliary variables present in Arya's approach. For example, enforcing that only one of the four instructions in the DG of Figure 6 can be executed at a time (at the pipeline stage called *EX*) is captured by the constraint *all-different*$(\{s(i_6, EX), s(i_7, EX), s(i_8, EX), s(i_9, EX)\})$. Ertl and Krall's experiments on a few programs for the Motorola 88100 processor [3] show that their approach improves 8.5% and 19% of the basic blocks scheduled by the Harris C compiler and GCC 1.3 [62], respectively. However, the richness of the resource model comes at the cost of low scalability: their experiments show that the approach cannot handle larger problems than Arya's scheduler despite the six-year gap between them.

Chou and Chung introduce an early combinatorial approach to instruction scheduling and bundling for multiple-issue processors [32]. The approach proposes a special-purpose enumeration technique based on the depth-first search exploration of a search tree where nodes correspond to bundles of instructions and edges correspond to bundle issues. The search tree is pruned by detecting and exploiting domination and equivalent relations among instructions. The detection analysis is an extension of early work on multiprocessor scheduling [135] where latencies are taken into account. Chou and Chung present a limited set of experiments where they optimally schedule random basic blocks of up to 20 instructions on processors with a bundle width of four instructions. The basic blocks are generated with different dependency densities (a density d means that d% of all instruction pairs are connected by a dependency). They find that the hardest basic blocks have a dependency density of around 12%; less (more) dependencies reduce the search by increasing the number of equivalent (dominated) instructions.

Inspired by early work on IP-based scheduling for high-level synthesis [64, 85], Leupers and Marwedel introduce an IP model that includes *alternative instruction versions* and *side-effect handling* for irregular, multiple-issue DSPs [111]. Alternative instruction versions (called *alternative encodings* by Leupers and Marwedel) support the selection of different instruction implementations, each using different resources for each input instruction. Side-effect handling supports inserting and scheduling *no-operation instructions* that inhibit writes to certain registers, and deals with pairs of instructions that must be scheduled together if a particular version of one of them is chosen. To schedule instructions in a basic block and take into account the particularities of DSPs, Leupers and Marwedel define two types of 0-1 variables: *scheduling variables* $s(v, i, k)$ ($v_{i,v,k}$ in the original notation) indicate whether version v of instruction i is issued in cycle k, and *no-operation variables* $n(r, k)$ indicate whether register r is inhibited by a no-operation in cycle k. The model includes constraints to ensure that each instruction is implemented and issued exactly once, dependencies are satisfied, instruction versions that use the same resource are not issued in the same cycle, and no-operations are inserted to prevent destroying live data stored in registers. This model can be seen as a generalization of Bowman's RCPSP approach [24]. The use of 0-1 scheduling variables for each cycle allows resource constraints to be expressed in a more direct and general form than in Arya's approach. However, the number of variables becomes dependent on an upper bound of the makespan, and modeling the dependency constraints becomes more complex: for each dependency (i, j) and cycle k in which i can be issued, a constraint is needed to enforce that if $s(v, i, k)$ holds, then some $s(v', j, l)$ must hold for l greater than k (such implications can be expressed in IP models with inequalities). For example (assuming that only one version v_1 is available for each instruction), the dependency (i_6, i_7) in Figure 6 yields the constraint for each cycle k that if $s(v_1, i_6, k)$ holds, then one of $s(v_1, i_7, k + 2), s(v_1, i_7, k + 3), \ldots$ must hold. Leupers and Marwedel present some experimental results on a few signal-processing basic blocks for

Texas Instruments' TMS320C2x [151] and Motorola DSP56k [100] processors, reporting optimal solutions for basic blocks of up to 45 instructions.

Modern approaches. Wilken et al. triggered a new research phase in combinatorial instruction scheduling [161]. This phase is characterized by a higher ambition on scalability where the goal is to solve the largest basic blocks (containing thousands of instructions) from established benchmark suites such as SPEC95 and SPEC2000 [147]. The focus shifts from being able to deal with highly irregular architectures to understanding and exploiting the structure of the DG (a key structure of instruction scheduling) and improving the underlying solving methods by applying problem-specific knowledge.

Wilken et al. use a simple IP model that can be seen as an application of Pritsker et al.'s RCPSP approach [132] targeting an ideal, single-issue processor, as seen in Table 3. The model contains scheduling variables $\mathrm{s}(i,k)$ (x_i^k in the original notation) indicating whether instruction i is issued in cycle k, and single-scheduling constraints, as in Leupers and Marwedel's approach. Following the RCPSP approach of Pritsker et al. yields more compact dependency constraints than in Leupers and Marwedel's model: if M is the maximum makespan of the schedule, then the issue cycle of an instruction i can be expressed as $\sum_{k=0}^{M} k \times \mathrm{s}(i,k)$, and a dependency (i,j) with latency l can be expressed as $\sum_{k=0}^{M} k \times \mathrm{s}(i,k) + l \leq \sum_{k=0}^{M} k \times \mathrm{s}(j,k)$. For example, the dependency (i_6, i_7) in Figure 6 yields the single constraint $\sum_{k=0}^{M} k \times \mathrm{s}(i_6,k) + 2 \leq \sum_{k=0}^{M} k \times \mathrm{s}(i_7,k)$.

The main contributions of Wilken et al. are DG transformations and methods to ease solving the IP problem. Wilken et al. assume single-issue processors although the model itself can be generalized for multiple-issue processors. The DG transformations include: *partitioning* to decompose DGs while preserving optimality, *redundant dependency elimination* to discard dependencies that do not affect the solution but incur overhead in the IP model (such as the branch dependency (i_6, i_9) in Figure 6), and *region linearization* to determine the order in which instructions from specific components of the DG, called *regions,* must appear in the optimal solution. Before solving, CP-style constraint propagation is applied to discard scheduling variables and reduce the size of the dependency constraints. During solving, violations of the dependency constraints are detected in the linear relaxations and corrected for all subsequent relaxations by adding additional constraints that are logically implied. Wilken et al.'s experiments on an ideal processor with latencies of up to three cycles show that their improvements enable scheduling all basic blocks (with up to 1,000 instructions) from the SPEC95 [147] floating-point benchmarks with optimal makespan.

Shortly after the paper of Wilken et al., Van Beek and Wilken introduced a CP approach targeting the same processor model [155]. The model contains finite integer scheduling variables $\mathrm{s}(i)$ (just i in the original notation) representing the issue cycle of each instruction i. Similarly to Ertl and Krall, dependency constraints are directly expressed as inequalities on the scheduling variables and an *all-different* global constraint is used to force a single issue in each cycle. The model also contains two types of *implied constraints* (logically redundant constraints that increase the amount of constraint propagation and thus reduce the search space): *distance constraints*, which impose a minimum issue distance among the boundary instructions of the regions, as defined by Wilken et al.; and *predecessor/successor constraints*, which enforce lower/upper bounds on the scheduling variable domains of instructions with multiple predecessors/successors (which can be seen as an application of *edge-finding*, a propagation algorithm for global scheduling constraints [156]). For example, instruction i_9 in the DG of Figure 6 yields the predecessor constraint $\mathrm{s}(i_9) \geq \min\{\mathrm{s}(i_7), \mathrm{s}(i_8)\} + 2$, which may improve propagation on the earliest possible issue cycle of i_9. Van Beek and Wilken demonstrate the effect of combining their implied constraints with a custom, lightweight CP solver: in their experiments, the CP approach is significantly faster (22 times) than the original IP approach for the same basic blocks from the SPEC95 floating-point benchmarks.

Heffernan and Wilken contributed two further DG transformations to ease solving: *superior nodes* (an extension to Chou and Chung's dominance relation) and *superior subgraphs* [77]. The first transformation identifies pairs of independent instructions with equal resource usage that can be ordered with artificial dependencies while preserving the optimality of the original scheduling problem. For example, i_8 in the DG from Figure 6 can always be scheduled before i_7 without loss of optimality, since i_8 is *closer* to its predecessor i_6 and i_7 is not *further away* from its successor i_9 in terms of shortest-path distance. Hence, an artificial dependency (i_8, i_7) with latency 0 can be safely added to the DG. The second transformation is a computationally heavier generalization that identifies pairs of isomorphic subgraphs in the DG such that their instructions can also be ordered while preserving optimality. The transformations are applied iteratively, as their outcome might expose further transformation opportunities. Heffernan and Wilken show that their transformations significantly reduce the upper and lower makespan bounds of the basic blocks in the SPEC2000 floating-point benchmarks, particularly for single-issue processors.

Malik et al. (including Van Beek) introduced a CP approach that combines and generalizes the ideas of Van Beek, Wilken, and Heffernan to multiple-issue processors while remaining scalable [115]. The model is an extension of Van Beek and Wilken's model where single-issue *all-different* constraints are replaced by the more general *global-cardinality* constraints [156]. A *global-cardinality* constraint for each processor resource limits the amount of instructions of the same type that can be issued in the same cycle. For instance, in Figure 6 a processor with a bundle width of two instructions yields the constraint *global-cardinality*$(\{\mathrm{s}(i_6), \mathrm{s}(i_7), \mathrm{s}(i_8), \mathrm{s}(i_9)\}, 2)$, limiting the number of scheduling variables that can be assigned the same cycle to two. These constraints reduce the search space using efficient propagation based on network flow algorithms and reduce solving overhead by eliminating the need to track the resource usage of each instruction with additional variables and constraints. Furthermore, the model contains four types of implied constraints: *distance* and *predecessor/successor constraints* generalized to multiple-issue processors; *safe pruning constraints*, which discard suboptimal schedules with idle cycles; and *superior subgraph constraints* (called *dominance constraints* by Malik et al.), which apply Heffernan and Wilken's transformations as constraint propagation. Unlike earlier modern approaches, Malik et al. experiment on a multiple-issue processor with a bundle width ranging from one to six and latencies similar to those of the PowerPC architecture [39]. Their results for the SPEC2000 benchmarks using a custom CP solver show that most basic blocks (up to 2,600 instructions) can be scheduled with provably optimal makespan. Interestingly, the experiments show that the number of basic blocks that are solved optimally in time does not decrease for bundle widths beyond two instructions.

4.2 Regional Instruction Scheduling

Local instruction scheduling is well understood and, despite its theoretical complexity, has been shown in practice to be feasible with combinatorial techniques. Unfortunately, though, the scope of local instruction scheduling severely limits the amount of instruction-level parallelism that can be exploited, particularly for control-intensive programs. To overcome this limitation, regional instruction scheduling approaches have been proposed that operate on multiple basic blocks with a certain CFG structure. A popular structure in the context of combinatorial instruction scheduling is the *superblock*. A superblock is a collection of consecutive basic blocks with a single entry point and possibly multiple exit points [86]. Figure 7(a) shows an example superblock with two basic blocks b_1 and b_2 annotated with their exit likelihoods.

Handling compensation code. Moving an instruction from one basic block to another often requires the insertion of *compensation code* into yet other basic blocks to preserve program semantics. In the case of superblocks, such a situation only arises when an instruction is moved after an exit

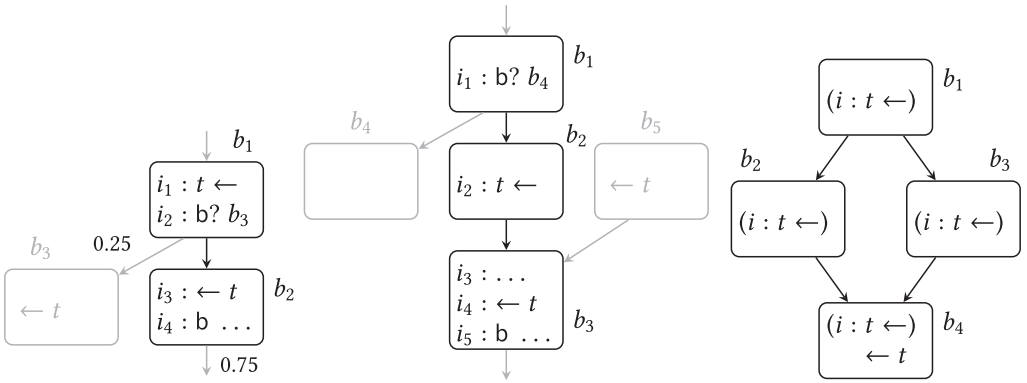


Fig. 7. Example CFGs (unrelated to the running example).

point where the result of the instruction must be available. For example, temporary t in Figure 7(a) must be available at the exit point of b_1, since it is used in b_3. Therefore, moving the instruction i_1 that defines t to b_2 must be compensated by the insertion of a duplicate of instruction i_1 in b_3. Most regional instruction scheduling approaches based on combinatorial optimization avoid handling compensation code by either disregarding the additional cost of placements of instructions that require compensation code or just disallowing such placements. The latter can be achieved by adding dependencies from such instructions to their corresponding exit points, at the expense of limiting the scope for optimization.

Superblock scheduling. Shobaki and Wilken presented the first combinatorial approach to superblock scheduling [144]. The approach introduces the foundations of a special-purpose enumeration technique that is later extended in multiple directions [143, 145]. The processor model is similar to that of Heffernan and Wilken [77] and Malik et al. [115] for local instruction scheduling. The key difference to local scheduling lies in the objective function: while local scheduling aims at minimizing the makespan of a basic block, the objective function in superblock scheduling minimizes the *weighted makespan*. The weighted makespan of a superblock is the sum of the cycles in which each exit instruction is scheduled weighted by their exit likelihood: $\sum_{b \in B} \text{weight}(b) \times \text{s}(\text{exit}(b))$, where B is the set of basic blocks in the superblock, $\text{exit}(b)$ gives the exit instruction of basic block b, and $\text{s}(i)$ gives the issue cycle of instruction i. For example, the objective function of the superblock from Figure 7(a) is $0.25 \times \text{s}(i_2) + 0.75 \times \text{s}(i_4)$. This objective function is common to all available combinatorial optimization approaches for superblock scheduling.

The enumeration technique follows a solving scheme akin to CP: it combines depth-first search with constraint propagation (called *pruning* by the authors) to reduce the search space. The technique searches for schedules with cost smaller than a certain upper bound and incrementally increases the upper bound until a schedule is found. The incrementality of the process and the completeness of the search guarantee that the found schedule is optimal. Each node in the search tree corresponds to an instruction to be scheduled next, and each root-to-leaf path corresponds to a complete schedule. At each non-leaf node, four propagation algorithms (called *tests* by Shobaki and Wilken) are applied to discard subtrees that cannot yield a schedule with cost smaller or equal than the upper bound: (1) testing whether the partial schedule up to the node is dominated according to the relation defined by Chou and Chung (see Section 4.1), (2) checking whether all instructions can be scheduled after simulating the propagation of dependency constraints, (3) comparing the partial

schedule to *similar* schedules seen before during search likewise Kessler [96] (see Appendix A), (4) and applying branch-and-bound with the cost lower bound obtained from a relaxation of the problem [108, 138]. If any of the algorithms finds that the partial schedule cannot be completed, then backtracking is applied.

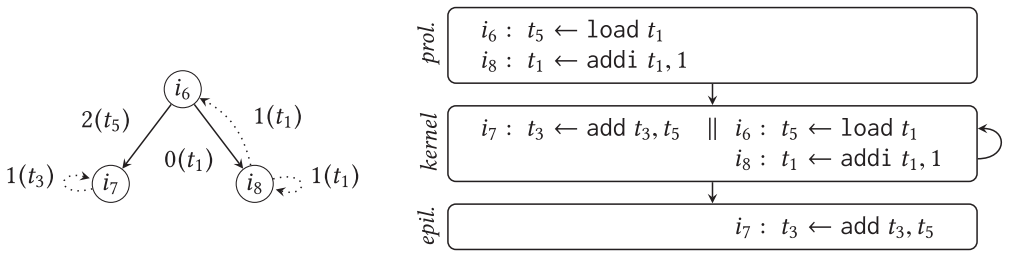
Shobaki and Wilken experiment with superblocks from SPEC2000 on different versions of an ideal processor with bundle width ranging from one to six instructions. Their results show that the approach improves the weighted makespan of the solutions generated by the best available heuristics at the time by 2.7% on average using a modest time limit of 1s per superblock.

Malik et al. present a CP approach that extends their local scheduler (discussed in Section 4.1) to superblocks and a richer resource model [114]. The basic model adds two elements to the local scheduling one: the objective function for superblocks, as in Shobaki and Wilken's approach, and additional variables and constraints to model two-dimensional resource usage and instructions that use all resources in their issue cycle. To scale to larger problem sizes, the model also incorporates superblock adaptations of *distance* and *superior subgraph constraints*. Malik et al. employ a more sophisticated solving procedure than in previous approaches, involving: *portfolio search*, where the level and complexity of constraint propagation is increased as solving proceeds; *impact-based search*, where the solver first selects the variable that causes most propagation to try different search alternatives on; and *decomposition*, where the independent basic blocks that belong to larger superblocks are solved separately. An overview of these techniques is provided by Van Beek [154] and Gomes and Walsh [65]. Malik et al. present a thorough (though static) experimental evaluation on the SPEC2000 [147] benchmarks for different variations of the PowerPC [39] processor. Their results show that the approach improves the makespan obtained from heuristic algorithms and conserves the scalability demonstrated for basic blocks, despite the generalization to superblocks and the incorporation of a richer resource model.

Beg and Van Beek integrate the superblock approach of Malik et al. with *spatial scheduling* [18] using CP. Spatial scheduling assigns instructions to clusters of functional units and register banks in clustered processors that are common in embedded systems. Spatial and instruction scheduling are interdependent, as assigning data-dependent instructions to different clusters requires scheduling additional intercluster copy instructions. The problem is solved by decomposition, where clusters are first assigned and for each given assignment an optimal schedule is computed as in Malik et al. Experiments by Beg and Van Beek on SPEC2000 benchmarks and different configurations of an ideal clustered processor based on PowerPC [39] show that integrating superblock and spatial scheduling can significantly improve the weighted makespan of heuristic approaches for clustered processors (by up to 26% on average, depending on the configuration) at the expense of compilation scalability (the largest superblock solved optimally contains 100 instructions compared to the 2,600 instructions of Malik et al.).

Trace scheduling. Superblocks can be generalized into *traces* to further increase the scope for scheduling, at the expense of higher complexity in handling compensation code. A trace is a collection of consecutive basic blocks with possibly multiple entry and exit points [58]. Figure 7(b) shows a trace with three basic blocks b_1, b_2, and b_3. Trace scheduling considers all possible execution paths through a trace. The example trace consists of three paths: $p_1 = (b_1)$, $p_2 = (b_1, b_2, b_3)$, and $p_3 = (b_3)$. Besides requiring compensation code due to exits (as with superblock scheduling), trace scheduling must insert compensation code when moving instructions before entry points. For example, i_4 in Figure 7(b) is executed by p_2 and p_3. Therefore, moving i_4 to b_2 (above the entry point of p_3) must be compensated by the insertion of a duplicate of i_4 in b_5.

The only available combinatorial approach to trace scheduling is an extension [145] of Shobaki and Wilken's superblock scheduler. The main difference with the superblock approach lies in the



(a) DG of b_2 with loop-carried dependencies and no control flow

(b) Output of software pipelining for b_2

Fig. 8. Software pipelining applied to the running example.

objective function, which minimizes a combination of the weighted makespan of all possible execution paths and the cost of inserting compensation code outside the trace: $kI + \sum_{p \in P} \text{weight}(p) \times (\text{s}(\text{exit}(p)) - \text{s}(\text{entry}(p)))$, where I is a variable indicating the number of compensation instructions introduced in external blocks, k is a constant factor to control the trade-off between trace speed and size of the rest of the function, and P is the set of paths from some entry to some exit in the trace. For example, assuming that the weights of p_1, p_2, and p_3 are 0.3, 0.5, and 0.2, respectively, the objective function of the trace from Figure 7(b) is $kI + 0.3 \times (\text{s}(i_1) - 0) + 0.5 \times (\text{s}(i_5) - 0) + 0.2 \times (\text{s}(i_5) - \text{s}(i_3))$. Shobaki et al.'s experimental results for the SPEC2006 integer benchmarks and an UltraSPARC IV [72] processor (with an instruction bundle width of four instructions) show that 91% of the traces (of up to 424 instructions) can be solved optimally with a 1s timeout, improving the weighted makespan slightly (2.7% on average) and reducing the compensation code noticeably (15% on average) compared to heuristic solutions.

Software pipelining. A fundamentally different form of regional scheduling is *software pipelining* [107, 137]. Software pipelining schedules the instructions of multiple iterations of a loop body simultaneously to increase the amount of available instruction-level parallelism. Typically, software pipelining is applied to inner-most loops consisting of single basic blocks, although extensions are proposed to deal with larger structures [139, Section 20.4]. Similarly to other instruction scheduling approaches, the input is represented by a DG, which in the case of software pipelining is extended with dependencies across iterations (called *loop-carried dependencies*). Figure 8(a) shows the DG of Figure 6 extended with loop-carried dependencies as dotted arcs. Instruction i_9 is omitted for simplicity (correct handling of control instructions needs extra care in software pipelining).

The output of software pipelining is a loop kernel with as short a makespan as possible (called *initiation interval* in this context), together with a prologue and epilogue to the kernel where the instructions of the initial and final iterations of the loop are scheduled. Common to all reviewed software pipelining approaches is the construction of schedules by successively increasing the length of the initiation interval and invoking the underlying optimization technique. The kernel of the first schedule found is thus guaranteed to be of minimum makespan. Most approaches define secondary objective functions to complement initiation interval minimization; usual candidates are register pressure (see Appendix A) and resource usage minimization. Figure 8(b) shows the output of software pipelining for the DG from Figure 8(a) and a multiple-issue processor. The kernel is formed with the overlap of two iterations so the sum of the array element (i_7) can be issued in parallel (||) with the load of the next element (i_6). This improves the makespan of the example loop kernel by one cycle compared to the non-pipelined version. A comprehensive review of the problem is given by Allan et al. [2].

The basic combinatorial approach to software pipelining, based on IP, is due to Govindarajan et al. [69]. The model includes two main groups of variables: *kernel scheduling variables* $\mathrm{a}(i,k)$ ($a_{k,i}$ in the original notation) indicate whether instruction i is scheduled in cycle k within the loop kernel, and *offset scheduling variables* $\mathrm{t}(i)$ give the cycle in which the first instance of instruction i is issued. The example from Figure 8(b) corresponds to a solution where: $\mathrm{a}(i_6,0)$, $\mathrm{a}(i_7,0)$ and $\mathrm{a}(i_8,1)$ are set to one; the remaining $\mathrm{a}(i,k)$ are set to zero; and $\mathrm{t}(i_6)=0$, $\mathrm{t}(i_7)=2$, and $\mathrm{t}(i_8)=1$. These variables are subject to linear constraints to enforce that: only one instance of each instruction is scheduled within the loop kernel, the capacity of the resources is not exceeded in any cycle, the data dependencies among instructions are satisfied, and the cycle of an instruction i in the kernel given by $\mathrm{a}(i,k)$ corresponds to its offset cycle $\mathrm{t}(i)$. Govindarajan et al.'s approach is processor resource-centric: the model minimizes resource usage as a secondary objective function and deals with rich resource models including two-dimensional resources and multiple resource usage. Their experiments on 22 loops from Livermore [117], SPEC92 [147], and Whetstone [36] benchmarks for a multiple-issue processor with a bundle width varying from one to six show that the optimal solution can be found in reasonable time (less than 30s) for 91% of the loops. Although the maximum size of a loop is not indicated, the reported initiation intervals suggest that the approach can handle loops in the order of tens of instructions.

Altman et al. (including Govindarajan) extend the resource usage minimization model [4] to handle asynchronous resource usage by tracking each individual use of each instruction with 0-1 variables and relating the use to its instruction's issue. Altman et al.'s experiments on 1,066 loops from Livermore, SPEC92, Linpack [41], NAS [11], and Whetstone benchmarks for a variant of the PowerPC-604 [39] show that, despite a significant increase in the complexity of the resource model, their extended approach can solve 75% of the loops optimally within 18mins.

Most subsequent work on combinatorial software pipelining focuses on register pressure minimization (see Appendix A).

4.3 Global Instruction Scheduling

The ultimate scope of instruction scheduling is to consider whole functions simultaneously without making any assumption on the CFG structure. This is referred to as *global instruction scheduling* in this survey. Global instruction scheduling allows an instruction to be moved from its original basic block (called *home basic block*) to any of its predecessor or successor basic blocks (called *candidate basic blocks*). An exception is made for *non-speculative* instructions (for example, those that can trigger exceptions), whose candidate basic blocks are limited to avoid altering the semantics of the original program. Handling compensation code in global scheduling becomes a general problem in which multiple copies of a moved instruction might need to be placed in different blocks to compensate for its motion.

Winkel. The first combinatorial approach to global instruction scheduling that handles compensation code and does not make any significant assumption on the CFG structure is due to Winkel [164, 166]. Winkel's IP approach is highly ambitious in that it captures 15 different types of code motion proposed in previous literature. However, the basic model is relatively compact, although the complete model is composed of 11 types of variables and 35 types of constraints to deal with the idiosyncrasies of the targeted processor and advanced model extensions [166]. The key idea is to consider for each basic block b and each original instruction i an optional copy of the instruction, and to enforce with linear constraints that one such copy is scheduled in every CFG path through the home basic block of i. The basic types of variables are: *scheduling variables* $\mathrm{s}(i,k,b)$ ($x_i^{b,k}$ in the original notation) to indicate whether a copy of instruction i is scheduled in cycle k in basic block b, *global motion variables* $\mathrm{c}(i,b)$ ($a_i^{\uparrow b}$ in the original notation) to indicate whether a

copy of instruction i is scheduled on all program paths preceding basic block b, and *block makespan variables* $\mathrm{m}(b,k)$ (B_k^b in the original notation) to indicate whether basic block b has a makespan of k. The basic types of constraints enforce that: every path leading to the home basic block of each instruction i contains a scheduled copy of i, the capacity of the resources is not exceeded in any cycle, the data dependencies among instructions are satisfied, and the makespan of each basic block b is equal to the latest cycle of the instructions scheduled in b. Additional variables and constraints are proposed, for example, to discard bundles containing incompatible groups of instructions [164] and to handle two-dimensional resource usage across basic block boundaries [166]. A detailed account of the extensions is given in Winkel's doctoral dissertation [165, Chapter 6]. The objective function is to minimize the weighted makespan of the function, which is the sum of the makespan of each basic block weighed by its estimated execution frequency. A second solving step is proposed where the makespan of each basic block is fixed and the amount of code motion is minimized with the intention to reduce register pressure. Figure 7(c) shows an example CFG with four basic blocks b_1, b_2, b_3, and b_4. b_4 is the home basic block of the original instruction i that has been expanded into one optional copy (parenthesized) per basic block. The result t is later used in b_4. The motion of i to b_2 is modeled by scheduling i in b_2 (that is, setting $\mathrm{s}(i,k,b_2)$ to one for some cycle k). Given that decision, the model constraints force the scheduling of i also in b_3 to compensate the motion. If i is instead scheduled in b_1 or in b_4, then every path leading to b_4 contains a copy of i and no other copy of i is scheduled.

Winkel's approach solves the generated IP problems with a commercial IP solver. Winkel demonstrates analytically that the model yields problems that can be solved efficiently with IP techniques, because their LP relaxations provide very tight bounds. This is achieved by proving that different relaxations of the model (for example, disregarding resource-capacity constraints) have LP relaxations with guaranteed integral solutions [165, Chapter 5]. Winkel confirms the analytic results with a dynamic experimental evaluation on the 104 functions of up to 600 instructions that account for 90% of the execution time in the SPEC2006 [147] benchmarks. His experiments for the Itanium 2 [118] processor (with a bundle width of six instructions) show that the *efficiency* of the IP model makes it feasible to solve functions with hundreds of instructions, yielding a runtime speedup of 10% compared to a heuristic global scheduler and 91% compared to optimal local instruction scheduling. Although Winkel's model does not directly capture the notion of register pressure, the second solving step is shown to be an effective replacement, as it reduces the percentage of spill code from the heuristic global scheduler's 0.8% to 0.2%.

4.4 Discussion

This section has reviewed combinatorial approaches to instruction scheduling proposed within the past 30 years. For local and superblock instruction scheduling, cross-fertilization from different combinatorial techniques has resulted in CP approaches that can optimally solve problems of thousands of instructions in seconds. The focus has been on handling in-order processors, and combinatorial models are available that capture many of the features of complex pipelines for such processors. For larger scopes, state-of-the-art approaches based on IP and special-purpose enumeration are able to deliver optimal solutions for medium-sized problems of hundreds of instructions.

Similarly to register allocation, combinatorial instruction scheduling has not seen much adoption in general-purpose production compilers yet. The solution-quality improvements reported for local and superblock approaches tend to be modest and, as a consequence of targeting in-order processors, only statically evaluated. A challenge for these approaches is to increase the currently limited range of supported processors by refining and dynamically evaluating the underlying combinatorial models. The remaining regional and global approaches show higher potential to improve code quality, but they do not scale beyond medium-sized problems yet, which limits their

Table 4. Integrated Register Allocation and Instruction Scheduling Approaches

approach	TC	SC	SP	RA	CO	LO	RP	LS	RM	MB	MA	BD	MU	2D	AS	SZ	DE
PROPAN	IP	superbl.	○	●	○	○	○	○	○	●	○	●	●	●	○	39	○
UNISON	CP	global	●	●	●	●	●	●	○	●	●	●	●	●	○	605	○
Chang 1997	IP	local	●	○	○	●	○	○	○	○	○	●	●	○	○	~10	○
Nagar. 2007	IP	sw. pip.	●	●	○	●	○	●	○	○	●	●	○	○	○	?	○

Technique (TC), scope (SC), spilling (SP), register assignment (RA), coalescing (CO), load-store optimization (LO), register packing (RP), live-range splitting (LS), rematerialization (RM), multiple register banks (MB), multiple allocation (MA), bundling (BD), multiple usage (MU), two-dimensional usage (2D), asynchronous usage (AS), size of largest problem solved optimally (SZ) in number of instructions, and whether a dynamic evaluation is available (DE).

applicability. These approaches are almost exclusively based on IP and special-purpose enumeration and might benefit from cross-fertilization with other techniques, as in the local case. Also, as the processor-memory gap increases, register-pressure-aware approaches (reviewed in Appendix A) could become increasingly relevant. A particular challenge in this context is to capture the cost incurred by high register pressure more accurately without suffering from the low scalability that characterizes the integrated approaches from Section 5.

5 INTEGRATED REGISTER ALLOCATION AND INSTRUCTION SCHEDULING

As many approaches reviewed in Sections 3 and 4 illustrate, the tight interdependencies between register allocation and instruction scheduling call for integrated approaches where the trade-offs between these tasks can be accurately reflected. Combinatorial optimization eases this effort (at the expense of some scalability) by allowing the composition of models of different tasks into a single combinatorial model that reflects the interdependencies among the tasks and can be solved with the same methods as for each task in isolation. Consequently, multiple combinatorial approaches integrating register allocation and instruction scheduling have been proposed.

Outline. Table 4 classifies integrated register allocation and instruction scheduling approaches with information about their optimization technique, scope, problem coverage, approximate scalability, and evaluation method. Sections 5.1, 5.2, and 5.3 discuss these approaches in further detail. Section 5.4 closes with a summary of developments and challenges in integrated combinatorial approaches. Appendix B complements this section with a review of approaches that also incorporate instruction selection into the combinatorial model.

5.1 PROPAN

Kästner proposes an IP approach to integrated instruction scheduling and register assignment in *PROPAN*, a system to reoptimize assembly code [94]. Spilling and its associated subproblems are not handled by such systems (see Table 4), as the input programs originate from assembly code and thus have acceptable register pressure. PROPAN provides two alternative IP models: a *time-based model* and an *order-based model*. The *time-based model* captures only instruction scheduling and has a structure similar to other reviewed IP models such as that of Leupers and Marwedel in Section 4.1; hence, the discussion focuses here on the *order-based model*.

The *order-based model* uses general scheduling variables $s(i)$ (t_i in the original notation) for each instruction i and inequalities on pairs of scheduling variables to model dependency constraints, as in Arya's approach in Section 4.1. Unlike earlier IP-based instruction scheduling models, it captures resource constraints as a compact resource allocation subproblem based on flow networks [93, 95, 169]. A *resource flow graph* is defined for each resource l, where nodes represent

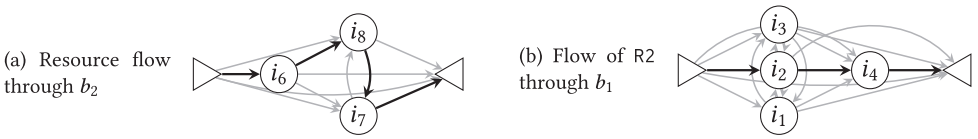


Fig. 9. Running example in PROPAN.

instructions (including source and sink nodes) and each arc (i, j) represents a potential *flow* of l through instructions i and j. Hence, a flow of l through a sequence of instructions represents successive use of l by these instructions and imposes a partial order among them. The flow of resource l through instructions i and j is reflected on the 0-1 variables $\mathrm{u}(l, i, j)$ ($x^l_{i,j}$ in the original notation). The partial order given by the $\mathrm{u}(l, i, j)$ variables is connected to the scheduling variables with additional constraints. Figure 9(a) shows the flow of a single-capacity resource through basic block b_2 from Figure 2(b), where instruction i_9 is omitted for simplicity. The resource flows through all instructions, enforcing a sequential schedule with the order (i_6, i_8, i_7).

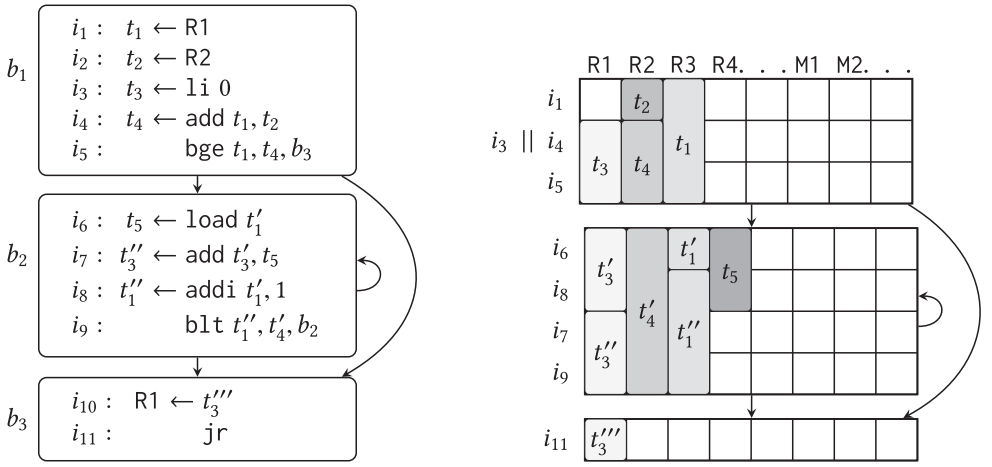
Register assignment is handled similarly to resource allocation: a *register flow graph* (similar to *reuse graphs* [21, 153]) is constructed where nodes correspond to instructions and each arc (i, j) corresponds to a potential *flow* of a register through instructions i and j. In this graph, a flow through a sequence of instructions implies that the temporaries defined by these instructions reuse a register. The flow of a register through instructions i and j is reflected on the 0-1 variables $\mathrm{r}(i, j)$ ($x^r_{i,j}$ in the original notation). Additional constraints ensure that two instructions i, j can only reuse a register if j is scheduled after the live range of the temporaries defined by i (similarly to how instruction-level parallelism is restricted by Malik [113] and Barany and Krall's [14] approaches discussed in Appendix A). Figure 9(b) shows a flow of register R2 through b_1 from Figure 2(b), where instruction i_5 is omitted for simplicity: first, R2 is assigned to t_2 by i_2; then, it is assigned to t_4 by i_4, which is possible since t_2 is last used at that point. The *register flow graph* can be seen as a simplified, dual version of the PRA's *multi-commodity network flow* from Section 3.1 where temporaries flow through registers.

The scope of PROPAN is the superblock [93, Chapter 7], and the objective function minimizes its makespan [93, Chapter 5]. Kästner evaluates the order-based model with integrated instruction scheduling and register assignment on five superblocks of up to 39 instructions from DSP applications for the ADSP-2106x processor [5]. In this setup, PROPAN generates code with almost 20% shorter makespan than list scheduling and optimal register assignment in isolation, but it does not seem to scale beyond small problems of up to 39 instructions [94].

5.2 UNISON

Castañeda Lozano et al. present *UNISON*, a CP approach that integrates multiple subproblems of register allocation with instruction scheduling [26]. Unlike most integrated combinatorial approaches, UNISON focuses on capturing a wide array of register allocation subproblems, including register assignment and packing, coalescing, and spilling for multiple register banks. The scope of UNISON's register allocation is global while instructions are only scheduled locally.

Castañeda Lozano et al. apply two program transformations before formulating a CP problem. First, the global register allocation problem is decomposed by renaming *global temporaries* (temporaries whose live ranges span several basic blocks) into one temporary per basic block, and relating the renamed temporaries across basic blocks similarly to SSA (see Section 3.4). Figure 10(a) shows the running example after the renaming. Renamed temporaries that originate from the same global temporary (for example t_1, t'_1, and t''_1) are defined and used at basic block boundaries (for example, t_1 is used at the exit of b_1 and t'_1 is defined at the entry of b_2), although such definitions and



(a) After renaming global temporaries (b) Instruction schedule and register assignment

Fig. 10. Running example in UNISON.

uses are omitted from the example for simplicity. Then, optional copy instructions are inserted at each definition and use of a temporary. These copies can be implemented by alternative instructions such as register-to-register moves, loads, and stores to support live-range splitting, spilling, and handling multiple register banks; or discarded to coalesce their corresponding temporaries. Memory locations are modeled as individual registers, yielding a compact model where the register assigned to a temporary determines its allocation (processor registers or memory), and spill code is reduced to simple copies between registers.

The CP model includes three main types of finite integer variables: *register variables* $\mathrm{r}(t)$ give the register to which temporary t is assigned, *instruction variables* $\mathrm{i}(i)$ give the instruction that implements copy i, and *scheduling variables* $\mathrm{s}(i)$ (c_i in the original notation) give the issue cycle of instruction i. Register assignment is modeled as a rectangle packing problem for each basic block using the *no-overlap* global constraint [19], which makes the number of variables in the model independent of the number of registers. In this subproblem, the live range of each temporary yields a rectangle to be packed into a grid where columns correspond to registers, similarly to the approach of Pereira and Palsberg [129]. Overlapping rectangles are forbidden, which models precisely that interfering temporaries cannot share registers. The integration between register allocation and instruction scheduling is reflected on the vertical dimension, where rows correspond to issue cycles and the top and bottom of temporary t's rectangle are determined by the issue cycles of t's definer and last user instruction. Figure 10(b) shows a solution to the rectangle packing problems (and hence a schedule and register assignment) for the example in Figure 10(a), assuming a multiple-issue processor with the latencies given by the DG of Figure 6. In this example, the rectangle corresponding to t_5 is packed in the column $\mathrm{r}(t_5) = \mathtt{R4}$ and stretches from row $\mathrm{s}(i_6) = 1$ to $\mathrm{s}(i_7) = 3$ (i_6 defines t_5 while i_7 uses it). The *registers* $\mathtt{M1}, \mathtt{M2}, \ldots$ correspond to memory locations, which are unused in the example, since there is no spilling (all copies are discarded). The packing subproblems from different basic blocks are connected by constraints that assign renamed temporaries that originate from the same global temporary (such as t_1, t'_1, and t''_1) to the same register. To ensure that the capacity of processor resources is not exceeded, UNISON uses *cumulative* global constraints [156]. Compared to Malik et al.'s approach based on *global-cardinality* constraints (see Section 4.1), the *cumulative* constraints enable a more compact model, as two-dimensional usage

can be modeled without resorting to additional variables and constraints. Finally, additional constraints are used to model the behavior of the copies, allocating the temporaries of a copy c to processor or memory *registers* according to the instruction $\mathrm{i}(c)$ that implements c, or forcing them to the same register if c is discarded. The objective is to minimize the *weighted cost* of the input function $\sum_{b \in B} \mathrm{weight}(b) \times \mathrm{cost}(b)$, where B is the set of basic blocks in the function and $\mathrm{cost}(b)$ is a generic cost function for basic block b. By combining different definitions of $\mathrm{weight}(b)$ and $\mathrm{cost}(b)$, the objective function can be adapted to optimize for different criteria. For example, setting $\mathrm{weight}(b)$ to the estimated execution frequency of b and $\mathrm{cost}(b)$ to b's makespan results in speed optimization, while setting $\mathrm{weight}(b)$ to one and $\mathrm{cost}(b)$ to the total size of the instructions in b results in code size optimization.

Castañeda Lozano et al. propose a model extension to capture load-store optimization and multiple allocation, and two methods to improve the robustness and scalability of UNISON [27]: *presolving* reformulates the CP problem by adding implied constraints and bounds; and *decomposition* splits the CP problem into a global problem (where the decisions affecting multiple basic blocks are taken), and a local problem per basic block b (where the internal decisions of b are taken independently).

Castañeda Lozano et al. evaluate UNISON statically with 55 functions of up to 1,000 instructions from MediaBench benchmarks [110] for Hexagon [33], a VLIW processor with a bundle width of four instructions and 32 general-purpose registers. The results show that UNISON reduces the weighted makespan of a heuristic approach (LLVM) by 7% on average, the approach can be easily adapted for code size optimization (resulting code size is on par with LLVM), and the extension with load-store optimization and multiple allocation indeed improves solution quality. The presolving and decomposition methods allow UNISON to scale up to functions of hundreds of instructions. An open question is whether instruction scheduling in UNISON could be extended to superblocks or beyond while preserving this scalability.

5.3 Other Approaches

Other IP approaches have been proposed that integrate multiple-issue instruction scheduling with subproblems of register allocation for different program scopes. Chang et al. propose an early IP model for local instruction scheduling and spilling that includes load-store optimization [30]. Similarly to UNISON, input programs are extended with optional store and load instructions after temporary definitions and before temporary uses. The model includes *scheduling variables* $\mathrm{s}(i,k)$ to indicate whether instruction i is scheduled in cycle k, *live range variables* $\mathrm{l}(t,k)$ to indicate whether temporary t is live in cycle k (similarly to Govindarajan et al.'s register-pressure-aware scheduling approach reviewed in Appendix A), and *activeness variables* $\mathrm{a}(i,k)$ to indicate whether the store or load instruction i is active in cycle k. These variables correspond to $x_{i,k}$, $U_{t,k}$, and $f_{i,k}$ in the original notation. Single-scheduling, dependency, and resource constraints are defined similarly to other time-based models in Section 4.1. The number of simultaneous live temporaries in each cycle is constrained to be less than or equal to the number of processor registers. The model only considers a store (load) instruction as active if it is not scheduled in the same cycle as its predecessor (successor) instruction in the DG. Two alternative objective functions are proposed: to minimize the used registers given a fixed makespan, and to minimize the makespan given a fixed number of processor registers. Chang et al. compare two manually specified models of a basic block with 10 instructions for an ideal processor, with and without spilling, and find that the model with spilling takes one order of magnitude more time to be solved optimally.

Nagarakatte and Govindarajan propose an IP model for register allocation and spill-code scheduling in software pipelining [120]. The approach assumes a software-pipelined loop kernel with a given initiation interval and computes (if feasible) a register allocation (including assignment,

spilling, load-store optimization, and multiple allocation) and a schedule of the spill code. The IP model includes three types of variables: *register assignment variables* $\mathrm{a}(t, r, k)$ indicate whether temporary t is assigned to register r in cycle k; *store variables* $\mathrm{s}(t, r, k)$ indicate whether temporary t is stored from register r to memory in cycle k; and *load variables* $\mathrm{l}(t, r, k)$ indicate whether temporary t is loaded from memory to register r in cycle k. These variables correspond to $TN_{t,r,k}$, $STN_{t,r,k}$, and $LTN_{t,r,k}$ in the original notation. The model has linear constraints to ensure that each temporary: is assigned to registers in the cycles in which it is defined and used; is assigned to a register in a certain cycle as a continuation of an assignment in previous cycles or if it is just loaded from memory; is assigned to a register before being stored to memory; and is stored in memory before being loaded into a register. Other constraints enforce that multiple temporaries are not assigned to the same register and that the capacity of the memory functional units is not exceeded. The model also includes dominance constraints: for instance, to enforce that each temporary is stored at most once and that temporaries whose values are already contained in registers are not loaded from memory. The objective function minimizes spill code. Nagarakatte and Govindarajan's experiments for an ideal processor with a bundle width of 10 instructions on 268 loops from the SPEC [147] and Perfect Club [20] benchmarks show that their approach yields 18% (13%) less spill code than its state-of-the-art heuristic counterpart [168], avoids increasing the initiation interval in 11% (12%) of the non-trivial loops, and takes on average 78 (18) seconds to solve each loop optimally for 32 (16) registers.

5.4 Discussion

This section has reviewed combinatorial approaches to integrated register allocation and instruction scheduling proposed within the past 20 years. As a consequence of their intrinsic complexity, the proposed IP and CP approaches only scale to small problems, except UNISON, which handles medium-sized problems. The improvement potential of the integrated approaches should theoretically be greater than the combined effect of solving each problem optimally in isolation; however, it is not fully determined yet due to scalability limitations (it has been observed that the improvement over heuristic approaches grows with the problem size) and the unavailability of dynamic evaluations.

The main challenge for integrated approaches is to scale up to larger problems: only then a thorough dynamic evaluation could reveal their improvement potential. Promising directions are identifying and exploiting problem decompositions as in UNISON and employing hybrid combinatorial optimization techniques [81, 83]. At the same time, improving on state-of-the-art heuristic approaches presupposes combinatorial models that cover the same subproblems at the same scope. None of the reviewed approaches has yet reached this level of coverage (for example, rematerialization is not addressed), and many of them are limited in scope (for example, performing register allocation locally). Overcoming these limitations while scaling to at least medium-sized problems remains a major challenge.

6 CONCLUSION

This article reviews and classifies the current body of literature on combinatorial register allocation and instruction scheduling, two central compiler back-end problems, thus allowing researchers and practitioners in the field to identify developments, trends, challenges, and opportunities.

Significant progress has been made for each of the two problems in isolation since the application of combinatorial optimization was first proposed in the 1980s. Today, the vast majority of register allocation and local instruction scheduling problems found in practice can be solved optimally on the order of seconds. Yet, combinatorial optimization is rarely applied in production compilers. Making combinatorial register allocation or local instruction scheduling more compelling involves

three challenges: reducing solving time, improving the quality of the generated code by capturing the cost of each optimization decision accurately in the combinatorial model, and demonstrating the improvement with thorough dynamic evaluations. Less fundamental factors such as licensing mismatches and software integration obstacles have to be dealt with as well.

More sophisticated instruction scheduling and integrated approaches have the potential to bring code quality improvements to motivate a paradigm shift; however, leveraging this potential presupposes combinatorial models that are accurate and complete enough to deliver substantial improvements in code quality. Devising such models and the corresponding solving methods to scale up to problem sizes of practical interest remains a major challenge for researchers in compilers and combinatorial optimization.

APPENDICES

A REGISTER-PRESSURE-AWARE INSTRUCTION SCHEDULING

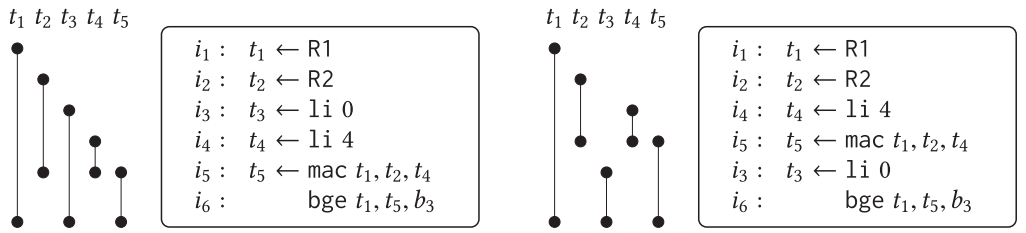
This appendix reviews register-pressure-aware (*RP*) instruction scheduling approaches. This task is performed before register allocation (see Figure 1) and aims at minimizing register pressure (number of temporaries that need to be stored simultaneously) or striking a balance with makespan minimization. The importance of each objective is typically dictated by the characteristics of the target: out-of-order processors tend to benefit from spill reduction by minimizing register pressure, while VLIW processors tend to benefit from higher instruction-level parallelism by minimizing makespan. When register pressure minimization is the only concern, instruction scheduling is typically reduced to its simplest configuration (*instruction ordering*, which assumes a single-issue processor with a trivial resource model and unit latency instructions, as reflected in Table 5 for the corresponding approaches). Addressing also instruction-level parallelism necessitates more advanced processor configurations. Register-pressure-aware approaches have shown moderate code quality improvements over heuristics on dynamic evaluations for out-of-order processors.

Figure 11 shows two instruction orderings of an extended version of b_1 from Figure 2(b) where the value held by t_5 is computed by a multiply-and-accumulate instruction (`mac`) that adds the base address of an array (t_1) with the product of its length (t_2) and its storage size (t_4). In Figure 11(a), four temporaries ($\{t_1, t_2, t_3, t_4\}$) need to be stored simultaneously in different registers between i_4 and i_5. In Figure 11(b), register-pressure-aware scheduling delays the issue of i_3 to achieve optimal register pressure with at most three temporaries simultaneously live (first $\{t_1, t_2, t_4\}$ and then $\{t_1, t_3, t_5\}$).

Table 5. Register-pressure-aware Instruction Scheduling Approaches

approach	TC	SC	BD	MU	2D	AS	SZ	DE
Kessler [96]	EN	local	○	○	○	○	25	○
Govindarajan et al. [71]	IP	local	○	○	○	○	~20	●
Malik [113]	CP	local	●	●	○	○	50	○
Shobaki et al. [143]	EN	local	○	○	○	○	664	●
Govindarajan et al. [70]	IP	sw. pipelining	●	●	○	○	?	○
Eichenberger and Davidson [49]	IP	sw. pipelining	●	●	●	●	41	○
Dupont de Dinechin [42]	IP	sw. pipelining	●	●	●	●	?	○
Domagała et al. [40]	CP	loop unrolling	○	○	○	○	?	●
Barany and Krall [14]	IP	global	○	○	○	○	~1,000	●

Technique (TC), scope (SC where EN stands for *enumeration*), bundling (BD), multiple usage (MU), two-dimensional usage (2D), asynchronous usage (AS), size of largest problem solved optimally (SZ) in number of instructions, and whether a dynamic evaluation is available (DE).



(a) Initial ordering (register pressure: 4) (b) Optimal ordering (register pressure: 3)

Fig. 11. Register-pressure-aware ordering for an extended version of b_1.

Outline. Table 5 classifies register-pressure-aware instruction scheduling approaches with information about their optimization technique, scope, problem coverage, approximate scalability, and evaluation method. As in Section 4, the discussion is structured by scope: Sections A.1, A.2, and A.3 cover local, regional, and global approaches.

A.1 Local RP Instruction Scheduling

The first combinatorial approach to local register-pressure-aware instruction scheduling is due to Kessler [96]. Kessler proposes a special-purpose enumeration technique for ordering and a scheduling extension with long instruction latencies. The technique explores a search tree where nodes correspond to sets of instructions that can be issued next (because their DG predecessors are already issued) and edges correspond to instruction issues. For example, the root node of the search tree for Figure 11 is $\{i_1, i_2, i_3, i_4\}$ and issuing one of i_1, i_2, i_3, and i_4 induces its children $\{i_2, i_3, i_4\}$, $\{i_1, i_3, i_4\}$, $\{i_1, i_2, i_4\}$, and $\{i_1, i_2, i_3\}$. The register pressure (called *register need* by Kessler) up to each node is kept during the search. A key improvement comes from the realization that nodes with the same sets of instructions can be merged in a dynamic programming fashion by simply selecting the one with lowest register pressure. This is possible, since the particular order in which earlier instructions are scheduled to arrive at a certain search node does not need to be known to compute the optimal solution. For example, the only information that must be kept at search node $\{i_1, i_2, i_4\}$ is that the lowest register pressure in any schedule including the predecessors of $\{i_1, i_2, i_4\}$ is three regardless of their particular order. Kessler experiments with both random and real basic blocks from Livermore [117] and Perfect Club [20] benchmarks. The instruction ordering approach manages to produce optimal solutions with a minimum register pressure of up to 13 simultaneously live temporaries for basic blocks of up to 51 instructions. An extension with long instruction latencies and any combination of makespan and register pressure minimization objectives is also proposed. Kessler's experiments show that handling multiple objectives (where makespan minimization is set to be the primary one) limits the scalability of the approach to basic blocks of up to 25 instructions.

Govindarajan et al. introduce the first IP model for register-pressure-aware instruction ordering [71]. The model is conceived to study the quality of a heuristic approach for spill reduction in out-of-order processors, hence sophisticated processor models for instruction-level parallelism are not considered. The IP model can be seen as a register-pressure-aware extension of Arya's with $\mathrm{s}(i)$ variables (f_i in the original notation) representing the issue cycle of each instruction i. Besides variables and constraints, as in Arya's approach (in its turn an application of Manne's RCPSP model [116]), the model includes *live range variables* $\mathrm{l}(t, k)$ ($s_{t,k}$ in the original notation) to indicate whether temporary t is live in cycle k and constraints to enforce this if $\mathrm{s}(d) \leq k$ and $\mathrm{s}(u) \geq k$ for the definer d and some user u of t. The objective function minimizes register pressure, which is the maximum sum of live temporaries among all cycles. Govindarajan et al. evaluate the

Table 6. Objectives of Local Register-pressure-aware Instruction Scheduling

approach	reg. pressure	makespan
Kessler [96]	min.	min.
Govindarajan et al. [71]	min.	-
Malik [113]	no excess	min.
Shobaki et al. [143]	min. excess	min.

quality of their heuristic approach by comparing with the results of the IP approach on 675 basic blocks from SPEC92, Linpack [41], Livermore [117], and NAS [11] benchmarks. Given the low scalability of their IP approach, only basic blocks of limited size (averaging 19 instructions) are considered. The comparison shows that Govindarajan et al.'s heuristic approach is near-optimal for these basic blocks: the IP approach only manages to slightly improve six of them. An open question is whether this conclusion can be extended to larger basic blocks: related studies [51, 115, 143, 161] suggest that the quality gap between heuristic and combinatorial approaches grows with the size of the problems. Govindarajan et al. also present a thorough evaluation of the impact of their heuristic approach on the out-of-order MIPS R10000 processor [167] for the SPEC95 floating-point benchmarks. Their experiments show that register-pressure-aware scheduling significantly reduces the number of executed stores and loads (by 7% and 11% on a processor configuration with 16 and 32 floating-point registers), and that this reduction yields a more modest speedup of around 3% and 4%. An open question is whether a higher speedup could be achieved by balancing register pressure minimization with instruction-level parallelism on a more refined processor model.

Govindarajan et al. introduce and exploit the notion of *lineages* (similar to *reuse graphs* [21, 153]) in their heuristic approach for register-pressure-aware instruction scheduling. A *lineage* is a path of data-dependent instructions in a basic block (for example, (i_2, i_5, i_6) in Figure 11). Lineages have the interesting property that all temporaries that connect them can reuse the same register. Multiple lineages can be fused using artificial data dependencies to control the register pressure of a basic block, at the expense of limiting instruction-level parallelism. Malik's doctoral dissertation [113, Chapter 5] explores the use of lineages in a combinatorial approach to find minimum-makespan schedules without *register pressure excess* (that is, not exceeding a given register pressure), as opposed to earlier approaches where register pressure is minimized, as shown in Table 6. Just eliminating register pressure excess assumes that, once the register pressure is equal to the number of processor registers, minimizing it further might only restrict instruction-level parallelism without providing any additional benefit. Since Malik's approach forbids register pressure excess, by construction it cannot handle basic blocks whose register pressure necessarily exceeds the number of processor registers. The dissertation shows that the DG transformations of Heffernan and Wilken [77] do not increase register pressure and that finding an instruction ordering with minimum register pressure is equivalent to finding a minimum cover of the DG by fusing lineages (where only the last instruction of a lineage is allowed to overlap with another lineage). Based on these insights, Malik proposes an approach that enumerates candidate fusions of lineages covering the DG and solves a minimum-makespan instruction scheduling problem for each candidate (as in earlier work by Malik et al.), iterating until the optimal solution is found. Malik's experiments with the same basic blocks and processor model as in his earlier work show that the overhead of addressing register pressure is significant: the approach can scale up to basic blocks of up to 50 instructions, two orders of magnitude less than the underlying CP approach for regular instruction scheduling. Comparing to Govindarajan et al.'s heuristic, Malik shows that reformulating the register-pressure-minimization problem as a makespan-minimization problem under register pressure constraints can improve the makespan of up to 8% of all basic blocks.

The latest register-pressure-aware instruction scheduling approach is due to Shobaki et al. [143]. The approach is based on a special-purpose enumeration technique developed previously by Shobaki and Wilken for regional instruction scheduling (see Section 4.2) that is extended to handle register pressure. Its major contribution is to improve the scalability of its related approaches by one order of magnitude (see Table 5). However, this comparison is complicated, since each approach optimizes according to different objectives: for Shobaki et al., the objective is to minimize a combination of register pressure excess (that is, likewise Malik's approach, only excessive register pressure is seen as undesirable) and makespan (see Table 6). As in previous approaches targeting out-of-order processors, the processor model is relatively simple, although long instruction latencies are considered for makespan minimization. The enumeration technique extends Shobaki and Wilken's depth-first search, constraint propagation-based technique (see Section 4.2) with register pressure-awareness. In particular, an additional propagation algorithm is introduced that evaluates the register pressure of a partial schedule to discard subtrees that cannot improve the best schedule found so far. Shobaki et al. are the first to experiment with a combinatorial instruction scheduling approach for Intel's x86, the most prominent out-of-order processor in the past two decades. Two configurations are considered: x86-32, which has very few registers, and x86-64, which includes an additional set of registers. Their experiments use the functions that take most execution time for each SPEC2006 benchmark and use the register-pressure-aware instruction scheduling heuristics in LLVM 2.9 [109] as a baseline. The results show that optimal solutions can be obtained for medium-sized basic blocks of hundreds of instructions. The runtime speedup results are consistent with those of Govindarajan et al., who also compare to a heuristic approach on an out-out-order processor: a modest speedup of 2.4% is obtained for the floating-point benchmarks on x86-64 while the speedup for the integer benchmarks is nearly negligible. Additionally, Shobaki et al. find that balancing register pressure with makespan minimization indeed has some impact. In particular, improving the x86 processor model with more precise latencies [60] gives an additional 0.4% speedup.

A.2 Regional RP Instruction Scheduling

Software pipelining. Govindarajan et al. present an alternative version of their basic software pipelining approach (see Section 4.2) that minimizes register pressure as a secondary goal [70] instead of focusing on rich resource modeling. The approach approximates register pressure by the amount of register *buffers* needed to store the values defined by each instruction in multiple iterations. Their experiments on 27 loops from Livermore, SPEC, Linpack [41], and Whetstone benchmarks for different configurations of a multiple-issue processor show that the optimal solution can be found in less than 5s for 87% of the loops and that in most cases the solutions found improve those of alternative heuristic approaches in either initiation interval, amount of register buffers, or both. Although a direct comparison is not available, the reported solving times suggest that the register-pressure-aware approach is slightly less scalable than its resource usage minimization counterpart.

Eichenberger and Davidson reformulate the dependency constraints in the basic IP model of Govindarajan et al. to ease solving [49]. The reformulation, based on earlier insights on the structure of IP scheduling models [31], makes the dependency constraints *0-1 structured*: each variable appears at most once and has only unit coefficients (see Table 1). This property improves the linear relaxation of the problem and thus reduces the amount of branch-and-bound search needed to solve it. Additionally, Eichenberger et al. propose a secondary objective function [50] that captures register pressure minimization more accurately than Govindarajan et al. Eichenberger and Davidson experiment with 1,327 loops from the Perfect Club [20], SPEC89, and Livermore benchmarks scheduled for the Cydra 5 processor [17] under different secondary objective functions. The

results show that the reformulation can reduce solving time by almost an order of magnitude and almost double the number of instructions that can be scheduled optimally in a given time budget, and that using the more accurate register pressure objective function instead of the register buffer minimization by Govindarajan et al. does not increase solving time.

Dupont De Dinechin introduced an alternative IP model that minimizes register pressure as accurately as Eichenberger et al. with a simpler structure but a potentially larger number of variables [42]. Unlike earlier IP approaches to software pipelining, Dupont De Dinechin's model has a single type of 0-1 *scheduling variables* $\mathrm{x}(i, k)$ indicating whether instruction i is scheduled in cycle k relative to the initial loop iteration. For example, Figure 8(b) corresponds to a solution where $\mathrm{x}(i_6, 0)$, $\mathrm{x}(i_7, 2)$, and $\mathrm{x}(i_8, 1)$ are set to one and the remaining $\mathrm{x}(i, k)$ are set to zero. Compared to the earlier approaches, Dupont De Dinechin's model features a better linear relaxation of the dependency constraints and is more amenable to heuristic approaches [23, 43] but suffers from a potentially larger number of variables, since the time horizon for the x variables is longer than for the former a variables. An implementation of Dupont De Dinechin's approach is available in the LAO code generator [44], which is a part of STMicroelectronics' ST200 [57] production compiler.

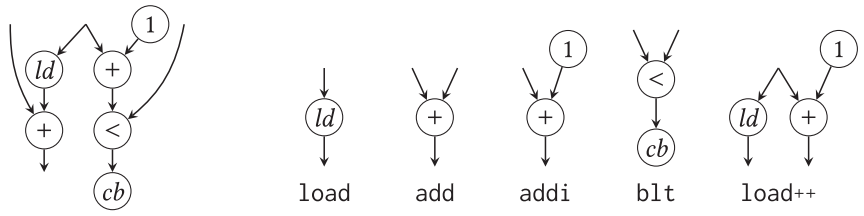
Loop unrolling. A simpler alternative with a similar goal to software pipelining is *loop unrolling*. Loop unrolling duplicates n iterations of a loop body (n is often called the *unrolling factor*) to increase the available instruction-level parallelism and reduce the overhead imposed by control instructions. Similarly to software pipelining, loop unrolling tends to increase register pressure and processor resource usage.

Domagała et al. presented a register-pressure-aware CP approach to integrated loop unrolling and instruction ordering where the loop body can be partitioned into *tiles* with different unrolling factors [40]. The approach proposes a sophisticated CP model to determine the order of each instruction, the partitioning of instructions into tiles, and the unrolling factor of each tile. The model follows Malik's local approach [113] in that it disallows excessive register pressure rather than minimizing it. The objective in Domagała et al.'s approach is to minimize the number of spills per iteration due to the structure of the resulting unrolling and tiling. Domagała et al.'s experiments show that about half of the high-register-pressure loops from the SPEC benchmarks can be solved optimally and that the overall number of spills is indeed reduced even though loop unrolling is performed.

A.3 Global RP Instruction Scheduling

Barany and Krall. While multiple combinatorial approaches to local and regional register pressure-aware instruction scheduling have been proposed, Barany and Krall are the first to propose a combinatorial approach that addresses the problem globally [14]. The strategy differs significantly from previous approaches: rather than directly finding a schedule that minimizes register pressure (or a combination with makespan minimization), Barany and Krall use IP to find code motion restrictions that might reduce register pressure by allowing pairs of temporaries to reuse registers. A code motion restriction is a set of conditions that allows two temporaries to reuse the same register, and it can be seen as a generalization of Govindarajan et al.'s notion of *lineage fusion* explained in Section A.1. Pairs of temporaries that might reuse the same register and their associated code motion restrictions are called *reuse candidates* and *blames* by Barany and Krall. Because the IP model does not have a global view of register pressure, the choice of restrictions that indeed reduce register pressure is delegated to a register allocator. After enforcing such restrictions, aggressive code motion can finally run with a reduced risk of increasing register pressure.

The IP model defining the set of code motion restrictions has three main types of variables: *restriction selection variables* $\mathrm{select}(r)$ to indicate whether code motion restriction r is selected,



(a) DFG of b_2 (b) Patterns for the example DFG

Fig. 12. Input to integrated code generation.

instruction placement variables place(i, b) to indicate whether instruction i can be placed in basic block b by the selected restrictions, and *instruction linearization variables* s(i) (instr(i) in the original notation), representing the order of instruction i within a linearization of the function's instructions, which Barany and Krall find necessary to avoid circular data and control dependencies. The linearization can be seen as a global instruction ordering. The model includes linear constraints to ensure that: data and control dependencies are reflected in the instruction linearization, all instructions can be placed in at least one basic block, instruction placements and linearizations are compatible with the selected restrictions, and the linearization of dependent instructions is compatible with their allowed placements. The objective function maximizes primarily the potential of the selected restrictions to decrease spill cost (by letting their corresponding temporaries reuse registers) and secondarily the number of allowed instruction placements (to preserve as much code motion freedom as possible). Unlike other instruction scheduling approaches, a solution does not directly correspond to a valid schedule but gives instead a set of code motion restrictions that might be useful to avoid costly spills. The final choice of restrictions is left to a heuristic PBQP register allocator (see Section 3.1), where a restriction is chosen if its associated temporaries are assigned to the same register. A small cost is associated with the choice to prevent unnecessary restrictions.

Barany and Krall use LLVM's [109] register-pressure-aware heuristic code motion and instruction scheduling algorithms as a baseline, and dynamically evaluate their approach with SPEC2000 functions of up to 1,000 instructions on an out-of-order ARM Cortex-A9 [8] processor. The results show that their approach does not have a notable impact on the execution time of the generated code: the code generated by LLVM is actually slightly slowed down (by 0.4%). Reducing the scope of the restrictions to single basic blocks improves to some degree the results, giving a speedup of 0.5% over LLVM. Barany and Krall conclude that the approach should be refined to capture the trade-offs between spilling and restricting certain code motions more accurately. An open question is whether this refinement is possible without a tighter integration between instruction scheduling and register allocation.

B INTEGRATED CODE GENERATION

This appendix reviews approaches that tackle one of the grand challenges of combinatorial code generation: the integration of instruction selection with combinatorial register allocation and instruction scheduling. Instruction selection transforms a program represented with intermediate operations into an equivalent one that uses instructions of a certain processor. Thus, the input program is represented at a higher level of abstraction, in terms of a *data-flow graph* (DFG) with intermediate *operations* and *data edges* (*edges* for short). Figure 12(a) shows the DFG corresponding to basic block b_2 in the running example. Operations *ld* and *cb* represent a load and a conditional branch. The immediate value 1 is represented as an operation with a single outgoing edge.

Table 7. Integrated Code Generation Approaches

approach	TC	SC	SP	RA	CO	LO	RP	LS	RM	MB	MA	BD	MU	2D	AS	SZ	DE
Wilson et al.[162]	IP	global	●	●	●	○	○	●	○	○	○	●	●	○	○	30	○
Gebotys [63]	IP	local	●	●	○	●	○	●	○	●	○	○	●	○	○	108	○
ICG	CP	local	●	●	●	●	○	●	○	●	○	●	○	○	○	23	○
OPTIMIST	IP	sw. pip.	●	○	○	●	○	●	○	●	●	●	●	●	●	100	○

Technique (TC), scope (SC), spilling (SP), register assignment (RA), coalescing (CO), load-store optimization (LO), register packing (RP), live-range splitting (LS), rematerialization (RM), multiple register banks (MB), multiple allocation (MA), bundling (BD), multiple usage (MU), two-dimensional usage (2D), asynchronous usage (AS), size of largest problem solved optimally (SZ) in number of instructions, and whether a dynamic evaluation is available (DE).

Instruction selection involves selecting among different *instruction patterns* (*patterns* for short) that can *cover* one or more nodes and edges of the DFG [80]. This problem interacts closely with register allocation and instruction scheduling, since the selection of processor instructions imposes constraints on the processor resources and register banks that are used. Figure 12(b) shows some patterns corresponding to MIPS32-like instructions. The pattern `load++` corresponds to a load instruction that increments the address after performing the memory access. Selecting the `load`, `add`, `addi`, and `blt` patterns results in the basic block b_2, shown in Figure 2(b) on page 7, while selecting the `load++` pattern removes the need for the `addi` instruction.

Outline. Table 7 classifies integrated code generation approaches with information about their optimization technique, scope, problem coverage, approximate scalability, and evaluation method. The rest of the appendix discuss these approaches in further detail.

Wilson et al. The first combinatorial approach to integrated code generation is proposed by Wilson et al. [162, 163]. The model is designed with the goal of generating code for DSPs with very limited and irregular register banks such as Motorola's DSP56k [100] processor. Remarkably, the IP model captures more register allocation subproblems than many approaches without instruction selection. Unfortunately, experimental results are not publicly available, which has limited the impact of the approach.

Wilson et al.'s IP model is rather compact in relation to the amount of captured subproblems. Both operations as well as edges of the DFG can be deactivated by the solver to model internal operations of complex processor instructions, optional spilling, and alternative implementations of certain operations such as computations of memory addresses. Unlike most studied approaches, the register allocation submodel is based on edges rather than temporaries, which increases the model granularity, since multiple uses of the same temporary yield different edges. Similarly to UNISON (see Section 5.2), Wilson et al. extend the input program with optional stores, loads, and register-to-register moves [163].

Similarly to Arya (Section 4.1), the model uses general integer variables $\mathrm{c}(o)$ (y_o in the original notation) representing the issue cycle of each operation o, which eases modeling dependency constraints but limits the generality of the resource model. The model is completed with the following types of 0-1 variables: *pattern selection variables* $\mathrm{s}(p)$ indicate whether pattern p is selected, *operation and edge activation variables* $\mathrm{a}(o)$ and $\mathrm{a}(e)$ indicate whether operation o and edge e are active, and *register assignment variables* $\mathrm{a}(e, r)$ indicate whether edge e is assigned to register r. These variables correspond to z_p, z_o, x_{ij}, and u_{ijr} in the original notation, where $e = (i, j)$. Linear constraints enforce that: pattern coverings do not overlap, edges and operations are active iff they are not internalized by a pattern covering, active edges are assigned to registers, edges related across basic blocks are assigned to the same register (similarly to UNISON), operations covered by the same pattern are scheduled in parallel, the definers of active edges are scheduled before their

users, active operations that use the same resource are not scheduled in parallel, and the definer and users of active edges that are assigned to the same register are scheduled such that the live ranges do not overlap. Additional constraints make it possible to model alternative implementations such that only one out of a group of operations appears in the generated code. For example, a solution for the basic block from Figure 12(a) where the pattern `load++` is selected is represented by setting the variables s(`add`), s(`blt`), and s(`load++`) to one. This forces the remaining s(p) variables to zero, since no pattern covering is allowed to overlap. In this solution, operations *ld*, 1, and + are scheduled in parallel, since they are covered by the `load++` pattern, which also deactivates the internalized edge from 1 to +. The objective function minimizes the maximum makespan across all basic blocks.

Unfortunately, Wilson et al. do not report detailed experimental results for their integrated code generation approach. The discussion included in the paper indicates that the approach is able to generate optimal solutions for up to 30 operations, of the same quality as hand-optimized code. Wilson et al. propose some techniques to scale up the approach, including taking preliminary decisions heuristically and decomposing functions into regions of basic blocks to be solved incrementally [163].

Gebotys. An alternative IP approach is due to Gebotys [63]. The approach solves *instruction compaction* as opposed to full instruction scheduling. Instruction compaction assumes a given fixed sequence of instructions and decides for each instruction i in the sequence whether i is scheduled in parallel with its successor. For example, instruction compaction for b_2 in Figure 2(b) assumes the sequence (i_6, i_7, i_8, i_9) and decides whether i_7 is scheduled in parallel with i_8 (the other compaction decisions are precluded by dependencies). Solving only compaction reduces the search space dramatically but limits the capacity to exploit instruction-level parallelism.

The model proposed by Gebotys is composed mostly of linear constraints derived from *Horn clauses*. A Horn clause is a logical formula with at most one conclusion [84]. The linear relaxation of an IP problem with only linear constraints derived from Horn clauses is the optimal solution to the IP problem. Even IP models not only composed of Horn constraints (as in Gebotys' model) become generally easier to solve, although they might still resort to branch-and-bound search.

Gebotys' model contains two types of 0-1 variables: *pattern covering variables* c(o, p) ($x_{o,p}$ in the original notation) indicate whether pattern p covers operation o, and *compaction variables* m(o) (p_o in the original notation) indicate whether operation o is compacted with its successor o' in the input instruction sequence, due to the same pattern covering both o and o'. For example, if the pattern `load++` is selected for Figure 12(a), then operations *ld*, 1, and + are compacted similarly to Wilson et al. Instead of using register assignment variables as in Wilson et al., Gebotys' model includes patterns for each different flow that a data edge could follow through registers and memory. Thus, selecting a pattern covering implicitly decides the register assignment and possible spilling of the covered edges. For example, the data edge between operations + and < in Figure 12(a) could follow the flow `R1` → `R1` (defined and used directly from `R1`), `R1` → `R2` (moved from `R1` to `R2`), `R1` → `M` → `R1` (defined in `R1`, spilled, and loaded into `R1`), and so on. This representation assumes that the target processor has a very small register file, as the number of patterns would otherwise explode. The model includes non-Horn constraints to enforce that pattern coverings do not overlap, and Horn constraints to enforce that: coverings are selected that assign interfering edges to different registers and operations are compacted whenever their edge assignments are compatible. The objective function is a weighted sum of three different costs: makespan, code size, and energy consumption (approximated as the number of memory accesses of the generated code).

Gebotys presents experimental results for six single-block signal processing examples and Texas Instruments' (TI) TMS320C2x [151]—a simple, single-issue DSP with very few registers. In the

experiments, the objective function is limited to minimizing the makespan, which in the case of TMS320C2x is directly proportional to code size. The results show a makespan improvement from 9% up to 118% relative to the code generated by TI's C compiler. Interestingly, the six basic blocks can be solved optimally from the first linear relaxation without resorting to branch-and-bound search. Gebotys also experiments with extending the model with complete instruction scheduling by adding a time dimension to the pattern selection variables and observes a dramatic increase of the solving time due to an explosion of variables in the model and the insufficiency of the initial linear relaxations to deliver the optimal solution. Since TMS320C2x is a single-issue processor, the extended model does not yield better solutions to any of the six single-block signal processing examples.

ICG. Bashford and Leupers propose *ICG*, a code generator that integrates instruction selection, register allocation, and instruction scheduling using an approach that is partially based on CP [16]. Unlike the other approaches, ICG decomposes the solving process into several stages, sacrificing global optimality in favor of solving speed. In Bashford and Leupers's paper, ICG is described as a procedural code generator with limited search capabilities. However, the model is presented in this survey as a monolithic CP model for clarity and consistency.

The ICG model has the following types of variables: *pattern covering variables* $\mathrm{v}(o)$ give the pattern that covers operation o, *register variables* $\mathrm{r}(o, op)$ give the register to which each use and definition operand op of operation o is assigned (register assignment in ICG is operand-based), *bundle type variables* $\mathrm{t}(o)$ give the type of bundle to which operation o belongs, *functional unit variables* $\mathrm{f}(o)$ give the functional unit used by operation o, and *operation cost variables* $\mathrm{c}(o)$ give the number of issue cycles required to execute operation o. These variables correspond to frt_o, $U_{o,op}$, and a triple $(t, f, c)_o$ in the original notation. The register assignment variables include special registers for memory banks and *internalized* operands covered by multi-operation patterns (corresponding to Wilson et al.'s deactivated edges). The model lacks scheduling variables, as instruction scheduling runs in a procedural fashion [16, Section 7]. The model contains constraints to ensure that: the registers assigned to operands of different operations are compatible; operations can only be bundled together if they have the same bundle type and do not use the same functional unit; and specific instruction constraints relating register assignments, bundle types, functional units, and costs within an operation are satisfied. The remaining usual constraints such as precedences among operations or non-interference among temporaries are handled by procedural algorithms with a limited form of search.

The solving procedure is split into three steps: first, a constraint problem is formulated and initial constraint propagation is performed to discard invalid registers for operands; then, a procedural algorithm schedules operations and generates spill code. The latter is performed *on-demand* by extending the problem with new variables and constraints corresponding to load, store, and register-to-register instructions. A certain level of search is permitted within each iteration, and reparation (where the problem is transformed into an easier one by means of, for example, spilling) is used as a last resort to handle search failures. The (implicit) goals are to minimize makespan and spill code. Finally, search is applied to assign values to all variables that have not been decided yet. A post-processing step is run to handle memory address computations.

Bashford and Leupers evaluate ICG on four basic blocks from DSPstone [158] benchmarks for the ADSP-210x [5] processor. The results show that the generated code has as short of a makespan as hand-optimized code and noticeably shorter than traditionally generated code. However, the scalability of ICG seems limited, as the time to generate code for a small basic block of 17 operations grows more than three times when the size is slightly increased to 23 operations.

OPTIMIST. The most recent approach to integrate code generation is *OPTIMIST* [53]. This project has explored the application of several search techniques for local code generation, including dynamic programming [98, 99] and genetic algorithms [54]. However, IP has established itself as the technique of choice within OPTIMIST as a natural yet formal method to extend the code generation model to more complex processors [54] and software pipelining [52, 53].

Compared to previous integrated approaches, the OPTIMIST IP model has a rich resource model in which, similarly to the early approach of Ertl and Krall presented in Section 4.1, the resource usage of an instruction is specified per pipeline stage. Another distinctive feature is that copies between different register banks and memory are only modeled implicitly by means of variables indicating their movement, rather than inserted in the input program in a preprocessing stage and defined as optional but otherwise regular operations. Similarly to UNISON (see Section 5.2), memory is modeled as a register bank, and thus spilling is unified with handling multiple register banks. Thus, the model captures register allocation but leaves out register assignment, which precludes coalescing and register packing, as seen in Table 7.

OPTIMIST's IP model is described first for local integrated code generation and extended later for software pipelining. The model (slightly simplified for clarity) includes the following types of 0-1 variables: *pattern selection variables* $\mathrm{s}(p,k)$ indicate whether pattern p is selected and issued in cycle k; *operation and edge covering variables* $\mathrm{c}(p,o,k)$ and $\mathrm{c}(p,e,k)$ indicate whether pattern p covers operation o and edge e in cycle k; *register allocation variables* $\mathrm{a}(t,b,k)$ indicate whether temporary t is allocated to register bank b in cycle k; and *copy variables* $\mathrm{x}(t,s,d,k)$ indicate whether temporary t is copied from source register bank s to destination register bank d in cycle k. These variables correspond to $s_{p,k}$, $c_{o,p,k}$, $w_{i,j,p,k}$ (where $e=(i,j)$), $r_{b,t,k}$, and $x_{t,s,d,k}$ in the original notation. For simplicity, the original pattern node and edge dimensions in the covering variables are omitted. The model's instruction selection constraints ensure that all operations are covered by exactly one pattern. The register allocation constraints ensure that the program temporaries: (1) are assigned to the register banks supported by their defining and using instructions in their definition and use cycles; (2) can only be allocated to a register bank b in a cycle k if they are defined in cycle k, are already allocated to b in cycle $k-1$, or are copied from another register bank in cycle k; (3) are not allocated to any register bank if they correspond to internalized edges; and (4) do not exceed the capacity of register banks in any cycle. These constraints are related to earlier IP models: (1) and (2) are similar to those of Nagarakatte and Govindarajan (see Section 5.3) and (3) is similar to those in Wilson et al.'s model. In both cases, the constraints are lifted from individual registers to register banks. Last, the model's instruction scheduling constraints ensure that: temporaries can only be used or copied after their definition cycle, and resources such as functional units are not overused in any cycle. The objective function, as is typical in local integrated approaches, minimizes the basic block's makespan.

For example, to generate the code corresponding to Figure 10(b) from the DFG shown in Figure 12(a), OPTIMIST sets the variables $\mathrm{s}(\texttt{load},0)$, $\mathrm{s}(\texttt{addi},1)$, $\mathrm{s}(\texttt{add},2)$, and $\mathrm{s}(\texttt{blt},3)$ to one and the remaining $\mathrm{s}(p,k)$ variables to zero. Since all temporaries in the example are allocated to the same register bank `R32` (containing registers `R1`, `R2`, ...), OPTIMIST sets the register variables $\mathrm{a}(t,\texttt{R32},k)$ to one for each temporary t and cycle k and which t is live, and all copy variables to zero.

The model extension to single-block software pipelining preserves the same variables and introduces only a few changes to handle two fundamental characteristics of software pipelining: the existence of loop-carried dependencies (see example in Figure 8(a)), and the fact that resource usage by an instruction and temporary live ranges might overlap over time across different iterations. As is common in combinatorial software pipelining, the initiation interval i is a fixed parameter and the IP problem is generated and solved for increasing values of i until the optimal solution is found.

Eriksson et al. present experimental results for both the local and the software pipelining IP approaches. The original experimental results are presented in two papers [53, 54]. This survey presents updated results from a rerun reported in Eriksson's doctoral dissertation [51], where a newer IP solver and host architecture are used. The local IP approach is compared to a heuristic code generator based on genetic algorithms. The input basic blocks are extracted from the *jpeg* and *mpeg2* benchmarks in MediaBench [110], and the targeted processor is a slightly modified version of TI's two-clustered TMS320C62x [152] processor. The local IP approach improves the solutions of the heuristic in around 50% of the cases, where the largest basic block solved optimally has 191 operations. In the original results published three years before, the largest basic block solved by the local IP approach had 52 operations, and only 40% of the basic blocks up to 191 operations could be solved [54]. This difference illustrates the speed with which solvers and architectures evolve to enable solving increasingly harder combinatorial problems.

Eriksson compares the fully integrated software pipelining IP approach to a separated approach where instruction selection is solved in isolation prior to register allocation and instruction scheduling. The experiments are run for the same processor as the local approach and loop kernels from SPEC2000, SPEC2006 [147], MediaBench, and FFmpeg benchmarks. The results show that the integrated approach improves the initiation interval for 39% of the kernels, yielding a total average improvement of around 8% [51, Chapter 8]. The approach scales up to kernels of approximately 100 operations [51, Chapter 4], 40 operations larger compared to the original experimental results [53].

ACKNOWLEDGMENTS

The authors are grateful for helpful comments from Magnus Boman, David Broman, Mats Carlsson, Mattias Eriksson, Lang Hames, Gabriel Hjort Blindell, Sven Mallach, Saranya Natarajan, Bernhard Scholz, and the anonymous reviewers.

REFERENCES

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall.

[2] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. 1995. Software pipelining. *Comput. Surveys* 27, 3 (Sept. 1995), 367–432.

[3] Mitch Alsup. 1990. Motorola's 88000 family architecture. *IEEE Micro* 10, 3 (May 1990), 48–66.

[4] Erik R. Altman, R. Govindarajan, and Guang R. Gao. 1995. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, 139–150.

[5] Analog Devices. 2017. ADSP-2106x SHARC Processor User's Manual. Retrieved from: http://www.analog.com/en/products/landing-pages/001/sharc-manuals.html.

[6] Andrew W. Appel and Lal George. 2000. Optimal Coalescing Challenge. Retrieved from: http://www.cs.princeton.edu/ appel/coalesce.

[7] Andrew W. Appel and Lal George. 2001. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 243–253.

[8] ARM. 2017. ARM Architecture Reference Manuals. Retrieved from: http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html.

[9] Christian Artigues, Sophie Demassey, and Emmanuel Neron. 2008. *Resource-constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. Wiley.

[10] Siamak Arya. 1985. An optimal instruction-scheduling model for a class of vector processors. *IEEE Trans. Comput.* C-34, 11 (Nov. 1985), 981–995.

[11] David H. Bailey and John T. Barton. 1985. *The NAS Kernel Benchmark Program*. Technical Report. NASA Ames Research Center, Mountain View, CA.

[12] Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. 2006. Constraint-based scheduling and planning. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 22, 671–800.

[13] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. 2001. *Constraint-based Scheduling*. Kluwer.

[14] Gergö Barany and Andreas Krall. 2013. Optimal and heuristic global code motion for minimal spilling. In *Compiler Construction (Lecture Notes in Computer Science)*, Vol. 7791. Springer, 21–40.
[15] 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 (Lecture Notes in Computer Science)*, Vol. 4382. Springer, 267–282.
[16] Steven Bashford and Rainer Leupers. 1999. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Auto. Embed. Syst.* 4 (Mar. 1999), 119–165.
[17] Gary R. Beck, David W. L. Yen, and Thomas L. Anderson. 1993. The Cydra 5 minisupercomputer: Architecture and implementation. *J. Supercomput.* 7, 1–2 (May 1993), 143–180.
[18] Mirza Beg and Peter van Beek. 2013. A constraint programming approach for integrated spatial and temporal scheduling for clustered architectures. *ACM Trans. Embed. Comput. Syst.* 13, 1 (Sept. 2013), 1–14.
[19] Nicolas Beldiceanu and Mats Carlsson. 2001. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In *Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Vol. 2239. Springer, 377–391.
[20] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi et al. 1988. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomput. Applic.* 3 (1988), 5–40.
[21] David A. Berson. 1996. *Unification of Register Allocation and Instruction Scheduling in Compilers for Fine-grain Parallel Architectures*. Ph.D. Dissertation. University of Pittsburgh, Pittsburgh, PA.
[22] Christian Bessiere. 2006. Constraint propagation. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 3, 29–84.
[23] Florent Blachot, Benoît Dupont de Dinechin, and Guillaume Huard. 2006. SCAN: A heuristic for near-optimal software pipelining. In *Parallel Processing (Lecture Notes in Computer Science*), Vol. 4128. Springer, 289–298.
[24] Edward H. Bowman. 1959. The schedule-sequencing problem. *Operations Research* 7, 5 (Sept. 1959), 621–624.
[25] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. 1991. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 122–131.
[26] 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 (Lecture Notes in Computer Science)*, Vol. 7514. Springer, 750–766.
[27] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2014. Combinatorial spill code optimization and ultimate coalescing. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 23–32.
[28] 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. Retrieved from: `arXiv:1409.7628`.
[29] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Comput. Lang.* 6, 1 (1981), 47–57.
[30] 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. *Comput. Math. Applic.* 34, 9 (Nov. 1997), 1–14.
[31] Samit Chaudhuri, Robert A. Walker, and John E. Mitchell. 1994. Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem. *IEEE Trans. Very Large Scale Integ. Syst.* 2, 4 (Dec. 1994), 456–471.
[32] Hong-Chich Chou and Chung-Ping Chung. 1995. An optimal instruction scheduler for superscalar processor. *IEEE Trans. Parallel Distrib. Syst.* 6, 3 (Mar. 1995), 303–313.
[33] 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 (Mar. 2014), 34–43.
[34] Quentin Colombet, Florian Brandner, and Alain Darte. 2015. Studying optimal spilling in the light of SSA. *ACM Trans. Archit. Code Optimiz.* 11, 4 (Jan. 2015), 1–26.
[35] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.
[36] Harold. J. Curnow and Brian A. Wichmann. 1976. A synthetic benchmark. *Comput. J.* 19, 1 (Feb. 1976), 43–49.
[37] 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 Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.
[38] William H. E. Day. 1970. Compiler assignment of data items to registers. *IBM Syst. J.* 9, 4 (Dec. 1970), 281–317.

[39] Keith Diefendorff and Ed Silha. 1994. The PowerPC user instruction set architecture. *IEEE Micro* 14, 5 (Oct. 1994), 30–41.
[40] Łukasz Domagała, Duco van Amstel, Fabrice Rastello, and P. Sadayappan. 2016. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the International Conference on Compiler Construction*. ACM, 143–151.
[41] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: Past, present and future. *Concur. Computat.: Pract. Exper.* 15, 9 (2003), 803–820.
[42] Benoît Dupont de Dinechin. 2004. From machine scheduling to VLIW instruction scheduling. *ST J. Res.* 1, 2 (Sep. 2004).
[43] Benoît Dupont de Dinechin. 2007. Time-indexed formulations and a large neighborhood search for the resource-constrained modulo scheduling problem. In *Proceedings of the Multidisciplinary International Conference on Scheduling: Theory and Applications*. MISTA, 144–151.
[44] Benoît Dupont de Dinechin, François de Ferrière, Christophe Guillon, and Arthur Stoutchinin. 2000. Code generator optimizations for the ST120 DSP-MCU core. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 93–102.
[45] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. 2009. Progressive spill code placement. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 77–86.
[46] Erik Eckstein. 2003. *Code Optimizations for Digital Signal Processors*. Ph.D. Dissertation. TU Wien, Austria.
[47] Erik Eckstein, Oliver König, and Bernhard Scholz. 2003. Code instruction selection based on SSA-graphs. In *Software and Compilers for Embedded Systems (Lecture Notes in Computer Science)*, Vol. 2826. Springer, 49–65.
[48] Erik Eckstein and Bernhard Scholz. 2003. Addressing mode selection. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 337–346.
[49] Alexandre E. Eichenberger and Edward S. Davidson. 1997. Efficient formulation for optimal modulo schedulers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 194–205.
[50] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. 1995. Optimum modulo schedules for minimum register requirements. In *Proceedings of the International Conference on Supercomputing*. ACM, 31–40.
[51] Mattias Eriksson. 2011. *Integrated Code Generation*. Ph.D. Dissertation. Linköping University, Sweden.
[52] Mattias Eriksson and Christoph Kessler. 2009. Integrated modulo scheduling for clustered VLIW architectures. In *High Performance Embedded Architectures and Compilers (Lecture Notes in Computer Science)*, Vol. 5409. Springer, 65–79.
[53] Mattias Eriksson and Christoph Kessler. 2012. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.* 11S, 1 (June 2012), 1–24.
[54] Mattias Eriksson, Oskar Skoog, and Christoph Kessler. 2008. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*. ACM, 11–20.
[55] Anton Ertl and Andreas Krall. 1991. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming (Lecture Notes in Computer Science)*, Vol. 528. Springer, 75–86.
[56] Heiko Falk, Norman Schmitz, and Florian Schmoll. 2011. WCET-aware register allocation based on integer-linear programming. In *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE, 13–22.
[57] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. 2000. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 203–213.
[58] Joseph A. Fisher. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.* 30, 7 (Jul. 1981), 478–490.
[59] Joseph A. Fisher. 1983. Very long instruction word architectures and the ELI-512. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 140–150.
[60] Agner Fog. 2017. *The Microarchitecture of Intel, AMD and VIA CPUs*. Technical Report. Technical University of Denmark.
[61] Changqing Fu and Kent Wilken. 2002. A faster optimal register allocator. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, 245–256.
[62] GCC2017. GCC, the GNU Compiler Collection. Retrieved from: https://gcc.gnu.org/.
[63] Catherine H. Gebotys. 1997. An efficient model for DSP code generation: Performance, code size, estimated energy. In *Proceedings of the International Symposium on System Synthesis*. IEEE, 41–47.
[64] Catherine H. Gebotys and Mohamed I. Elmasry. 1991. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *Proceedings of the Design Automation Conference*. ACM, 2–7.

[65] Carla Gomes and Toby Walsh. 2006. Randomness and structure. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 18, 639–664.
[66] James R. Goodman and Wei-Chung Hsu. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the International Conference on Supercomputing*. ACM, 442–452.
[67] David W. Goodwin and Kent Wilken. 1996. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw.: Pract. Exper.* 26, 8 (Aug. 1996), 929–965.
[68] R. Govindarajan. 2007. Instruction scheduling. In *The Compiler Design Handbook* (2nd ed.). CRC.
[69] R. Govindarajan, Erik R. Altman, and Guang R. Gao. 1994. A framework for resource-constrained rate-optimal software pipelining. In *Vector and Parallel Processing (Lecture Notes in Computer Science)*, Vol. 854. Springer, 640–651.
[70] R. Govindarajan, Erik R. Altman, and Guang R. Gao. 1994. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, 85–94.
[71] R. Govindarajan, Hongbo Yang, José Nelson Amaral, Chihong Zhang, and Guang R. Gao. 2003. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Trans. Comput.* 52, 1 (Jan. 2003), 4–20.
[72] D. Greenley, J. Bauman, D. Chang, D. Chen, R. Eltejaein, P. Ferolito, P. Fu, R. Garner, D. Greenhill, H. Grewal et al. 1995. UltraSPARC: The next generation superscalar 64-bit SPARC. In *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*. IEEE, 442–451.
[73] Daniel Grund and Sebastian Hack. 2007. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction (Lecture Notes in Computer Science)*, Vol. 4420. Springer, 111–125.
[74] Sebastian Hack, Daniel Grund, and Gerhard Goos. 2006. Register allocation for programs in SSA-form. In *Compiler Construction (Lecture Notes in Computer Science)*, Vol. 3923. Springer, 247–262.
[75] Lang Hames. 2011. *Specification Driven Register Allocation*. Ph.D. Dissertation. University of Sydney, Australia.
[76] Lang Hames and Bernhard Scholz. 2006. Nearly optimal register allocation with PBQP. In *Modular Programming Languages (Lecture Notes in Computer Science)*, Vol. 4228. Springer, 346–361.
[77] Mark Heffernan and Kent Wilken. 2006. Data-dependency graph transformations for instruction scheduling. *J. Sched.* 8, 5 (2006), 427–451.
[78] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.
[79] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. 2003. Graph coloring vs. optimal register allocation for optimizing compilers. In *Modular Programming Languages (Lecture Notes in Computer Science)*, Vol. 2789. Springer, 202–213.
[80] Gabriel Hjort Blindell. 2016. *Instruction Selection: Principles, Methods, and Applications*. Springer.
[81] John N. Hooker. 2006. Operations research methods in constraint programming. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 15, 527–570.
[82] John N. Hooker. 2012. *Integrated Methods for Optimization* (2nd ed.). Springer.
[83] Holger H. Hoos and Edward Tsang. 2006. Local search methods. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 5, 135–168.
[84] Alfred Horn. 1951. On sentences which are true of direct unions of algebras. *Symbol. Logic* 16, 1 (03 1951), 14–21.
[85] Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu. 1991. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. Comput.-Aided Design Integ. Circ. Syst.* 10, 4 (April 1991), 464–475.
[86] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab et al. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 1–2 (May 1993), 229–248.
[87] Infineon Technologies. 2017. Carmel. Retrieved from: https://www.infineon.com.
[88] Infineon Technologies. 2017. TriCore 1 Architecture Overview Handbook. Retrieved from: http://www.infineon.com/dgdl/TC1_3_ArchOverview_1.pdf?fileId=db3a304312bae05f0112be86204c0111.
[89] Intel Corporation. 2017. Intel 64 and IA-32 Architectures Software Developer Manuals. Retrieved from: https://software.intel.com/en-us/articles/intel-sdm.
[90] Joxan Jaffar and Jean-Louis Lassez. 1987. Constraint logic programming. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 111–119.
[91] Richard K. Johnsson. 1973. *A Survey of Register Allocation*. Technical Report. Carnegie Mellon University, Pittsburgh, PA.
[92] Gerry Kane. 1996. *PA-RISC 2.0 Architecture*. Prentice Hall.
[93] Daniel Kästner. 2000. *Retargetable Postpass Optimisation by Integer Linear Programming*. Ph.D. Dissertation. Saarland University, Germany.

[94] Daniel Kästner. 2001. PROPAN: A retargetable system for postpass optimisations and analyses. In *Languages, Compilers, Tools and Theory for Embedded Systems (Lecture Notes in Computer Science)*, Vol. 1985. Springer, 63–80.
[95] Daniel Kästner and Marc Langenbach. 1999. Code optimization by integer linear programming. In *Compiler Construction (Lecture Notes in Computer Science)*, Vol. 1575. Springer, 122–136.
[96] Christoph Kessler. 1998. Scheduling expression DAGs for minimal register need. *Comput. Lang.* 24, 1 (Sep. 1998), 33–53.
[97] Christoph Kessler. 2010. Compiling for VLIW DSPs. In *Handbook of Signal Processing Systems*. Springer, 603–638.
[98] Christoph Kessler and Andrzej Bednarski. 2001. A dynamic programming approach to optimal integrated code generation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 165–174.
[99] Christoph Kessler and Andrzej Bednarski. 2006. Optimal integrated code generation for VLIW architectures. *Concur. Computat.: Pract. Exper.* 18, 11 (2006), 1353–1390.
[100] Kevin L. Kloker. 1987. The architecture and applications of the Motorola DSP56000 digital signal processor family. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 523–526.
[101] David Koes and Seth Copen Goldstein. 2005. A progressive register allocator for irregular architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 269–280.
[102] David Koes and Seth Copen Goldstein. 2006. A global progressive register allocator. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 204–215.
[103] David Koes and Seth Copen Goldstein. 2009. Register allocation deconstructed. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*. ACM, 21–30.
[104] Timothy Kong and Kent Wilken. 1998. Precise register allocation for irregular architectures. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, 297–307.
[105] Tjalling C. Koopmans and Martin Beckmann. 1957. Assignment problems and the location of economic activities. *Econometrica* 25, 1 (1957), 53–76.
[106] Ulrich Kremer. 1997. Optimal and near-optimal solutions for hard compilation problems. *Parallel Proc. Lett.* 7, 4 (1997), 371–378.
[107] Monica Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 318–328.
[108] Michel Langevin and Eduard Cerny. 1996. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Design Auto. Electron. Syst.* 1, 4 (Oct. 1996), 443–455.
[109] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 75–88.
[110] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, 330–335.
[111] Rainer Leupers and Peter Marwedel. 1997. Time-constrained code compaction for DSP's. *IEEE Trans. Very Large Scale Integ. Syst.* 5, 1 (Mar. 1997), 112–122.
[112] Emilio Luque and Ana Ripoll. 1984. Integer linear programming for microprograms register allocation. *Inform. Proc. Lett.* 19, 2 (Sep. 1984), 81–85.
[113] Abid M. Malik. 2008. *Constraint Programming Techniques for Optimal Instruction Scheduling*. Ph.D. Dissertation. University of Waterloo, Canada.
[114] 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 (Lecture Notes in Computer Science)*, Vol. 5202. Springer, 97–111.
[115] Abid M. Malik, Jim McInnes, and Peter van Beek. 2008. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *Artific. Intell. Tools* 17, 1 (2008), 37–54.
[116] Alan S. Manne. 1960. On the job-shop scheduling problem. *Op. Res.* 8, 2 (Mar. 1960), 219–223.
[117] Francis H. McMahon. 1986. *Livermore Fortran Kernels: A Computer Test of Numerical Performance Range*. Technical Report. Lawrence Livermore National Laboratory, Livermore, CA.
[118] Cameron McNairy and Don Soltis. 2003. Itanium 2 processor microarchitecture. *IEEE Micro* 23, 2 (Mar. 2003), 44–55.
[119] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. 1995. *Combining Instruction Scheduling and Register Allocation*. Technical Report. Stanford University, Stanford, CA.
[120] 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 (Lecture Notes in Computer Science)*, Vol. 4420. Springer, 126–140.
[121] Mayur Naik and Jens Palsberg. 2002. Compiling with code-size constraints. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 120–129.

[122] V. Krishna Nandivada. 2007. Advances in register allocation techniques. In *The Compiler Design Handbook* (2nd ed.). CRC.
[123] V. Krishna Nandivada and Rajkishore Barik. 2013. Improved bitwidth-aware variable packing. *ACM Trans. Archit. Code Optimiz.* 10, 3 (Sep. 2013), 1–22.
[124] V. Krishna Nandivada and Jens Palsberg. 2006. SARA: Combining stack allocation and register allocation. In *Compiler Construction (Lecture Notes in Computer Science)*, Vol. 3923. Springer, 232–246.
[125] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. 2007. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis (Lecture Notes in Computer Science)*, Vol. 4634. Springer, 153–169.
[126] George L. Nemhauser and Laurence A. Wolsey. 1999. *Integer and Combinatorial Optimization*. Wiley.
[127] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. 2009. Propagation via lazy clause generation. *Constraints* 14, 3 (2009), 357–391.
[128] Fernando Magno Quintão Pereira. 2008. *A Survey on Register Allocation*. Technical Report. University of California-Los Angeles, Los Angeles, CA.
[129] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register allocation by puzzle solving. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 216–226.
[130] Shlomit S. Pinter. 1993. Register allocation with instruction scheduling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 248–257.
[131] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. 2009. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro* 29, 5 (Sep. 2009), 18–29.
[132] A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. 1969. Multiproject scheduling with limited resources: A zero-one programming approach. *Manag. Sci.* 16, 1 (Sep. 1969), 93–108.
[133] Jonathan Protzenko. 2009. *A Survey of Register Allocation Techniques*. Technical Report. École Polytechnique, France.
[134] Vaclav Rajlich and M. Drew Moshier. 1984. *A Survey of Algorithms for Register Allocation in Straight-line Programs*. Technical Report. University of Michigan, Ann Arbor, MI.
[135] Chittoor V. Ramamoorthy, K. Mani Chandy, and Mario J. Gonzalez. 1972. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. Comput.* 21, 2 (Feb. 1972), 137–146.
[136] B. Ramakrishna Rau and Joseph A. Fisher. 1993. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomput.* 7, 1–2 (May 1993), 9–50.
[137] B. Ramakrishna Rau and Christopher D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *ACM SIGMICRO Newslett.* 12, 4 (Dec. 1981), 183–198.
[138] Minjoong Rim and Rajiv Jain. 1994. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. Comput.-Aided Design* 13, 4 (1994), 451–458.
[139] Hongbo Rong and R. Govindarajan. 2007. Advances in software pipelining. In *The Compiler Design Handbook* (2nd ed.). CRC.
[140] Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). 2006. *Handbook of Constraint Programming*. Elsevier.
[141] Richard M. Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (Jan. 1978), 63–72.
[142] Bernhard Scholz and Erik Eckstein. 2002. Register allocation for irregular architectures. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, Tools and Theory for Embedded Systems*. ACM, 139–148.
[143] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Trans. Archit. Code Optimiz.* 10, 3 (Sep. 2013), 1–31.
[144] Ghassan Shobaki and Kent Wilken. 2004. Optimal superblock scheduling using enumeration. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, 283–293.
[145] Ghassan Shobaki, Kent Wilken, and Mark Heffernan. 2009. Optimal trace scheduling using enumeration. *ACM Trans. Archit. Code Optimiz.* 5, 4 (Mar. 2009), 1–32.
[146] Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 277–288.
[147] Standard Performance Evaluation Corporation. 2017. SPEC CPU Benchmarks. Retrieved from: https://www.spec.org/benchmarks.html.
[148] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 108–120.
[149] Dominic Sweetman. 2006. *See MIPS Run, Second Edition*. Morgan Kaufmann.
[150] Sriraman Tallam and Rajiv Gupta. 2003. Bitwidth aware global register allocation. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 85–96.

[151] Texas Instruments. 2017. TMS320C20x User's Guide. Retrieved from: http://www.ti.com/lit/ug/spru127c/spru127c.pdf.
[152] Texas Instruments. 2017. TMS320C62x DSP CPU and Instruction Set Reference Guide. Retrieved from: www.ti.com/lit/pdf/spru731.
[153] Sid Ahmed Ali Touati. 2002. *Register Pressure in Instruction Level Parallelism*. Ph.D. Dissertation. Versailles Saint-Quentin-en-Yvelines University, France.
[154] Peter van Beek. 2006. Backtracking search algorithms. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 4, 85–134.
[155] Peter van Beek and Kent Wilken. 2001. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Vol. 2239. Springer, 625–639.
[156] Willem-Jan van Hoeve and Irit Katriel. 2006. Global constraints. In *Handbook of Constraint Programming*, Francesca Rossi, Peter van Beek, and Toby Walsh (Eds.). Elsevier, chapter 6, 169–208.
[157] Robert J. Vanderbei. 2013. *Linear Programming: Foundations and Extensions*. Springer.
[158] Vojin Živojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. 1994. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the Conference on Signal Processing Applications and Technology*. DSP Associates, 715–720.
[159] Harvey M. Wagner. 1959. An integer linear-programming model for machine scheduling. *Naval Res. Logist. Quart.* 6, 2 (Jun. 1959), 131–140.
[160] Fredrik Wickberg and Mattias Eriksson. 2017. Outperforming state-of-the-art compilers in Unison. Ericsson research blog entry. Retrieved from: https://www.ericsson.com/research-blog/outperforming-state-art-compilers-unison/.
[161] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 121–133.
[162] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. 1994. An integrated approach to retargetable code generation. In *Proceedings of the International Symposium on High-Level Synthesis*. IEEE, 70–75.
[163] Tom Wilson, Gary Grewal, Shawn Henshall, and Dilip Banerji. 2002. An ILP-based approach to code generation. In *Code Generation for Embedded Processors (Engineering and Computer Science)*, Vol. 317. Springer, 103–118.
[164] Sebastian Winkel. 2004. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 189–200.
[165] Sebastian Winkel. 2004. *Optimal Global Instruction Scheduling for the Itanium Processor Architecture*. Ph.D. Dissertation. Saarland University, Germany.
[166] Sebastian Winkel. 2007. Optimal versus heuristic global code scheduling. In *Proceedings of the International Symposium on Microarchitecture*. IEEE, 43–55.
[167] Kenneth C. Yeager. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro* 16, 2 (Apr. 1996), 28–40.
[168] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. 2000. Improved spill code generation for software pipelined loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 134–144.
[169] Li Zhang. 1996. *Scheduling and Allocation with Integer Linear Programming*. Ph.D. Dissertation. Saarland University, Germany.
[170] Zilog, Inc. 2017. Z8 CPU User Manual. Retrieved from: http://www.zilog.com/docs/um0016.pdf.

Received December 2016; revised February 2018; accepted March 2018