Register Reuse Scheduling

Gergö Barany
Institute of Computer Languages
Vienna University of Technology
gergo@complang.tuwien.ac.at

ABSTRACT

The amount of spill code generated by a compiler backend has crucial effects on program performance. Instruction scheduling before register allocation may cause live range overlaps that lead to suboptimal spill code. Even when a local scheduler tries to minimize register usage, its results can leave room for improvement regarding overall spill costs.

We present Register Reuse Scheduling (RRS), a novel approach to global register allocation that derives a register assignment while minimizing spill code by locally (re-)ordering independent operations. Using basic block data dependence graphs (DDGs), we identify possibly interfering live ranges whose interferences may be avoided by appropriate scheduling of instructions: Sequencing of independent computations allows the reuse of registers to avoid spilling. Such scheduling decisions may be enforced by adding arcs to the DDG. We use global spill cost information to identify a set of profitable arcs that enable register reuse. The corresponding interferences need not be present in the register allocation problem. A standard near-optimal register allocator computes a register assignment. The register reuses that are implicit in the assignment allow us to select arcs to add to the DDG and construct a schedule with minimal spill costs.

We present experimental data that shows that our approach can significantly decrease the amount of spills as compared to a local scheduling heuristic aimed at minimizing register usage. On average, we spill 8.9 % fewer values and reduce static spill costs by 3.4 %.

1. INTRODUCTION

The phase of register allocation and spilling is one of the major parts of an optimizing compiler backend. Spilling and reloading registers incurs considerable costs because reading a value from memory is orders of magnitude slower than reading a value from a register.

Especially in the context of embedded systems, other costs of spilling are relevant as well: Memory accesses cost more energy than register accesses, shortening the battery life of mobile devices. More store and load instructions generated by the compiler also cause an increase in code size, which may increase the device's memory cost. A larger code size, in turn, can also lead to more memory accesses and instruction cache misses, further lowering the execution speed of the program and energy economy of the device.

This paper presents a new method aimed at minimizing spill code by attempting to find an instruction schedule that allows for the least expensive register allocation. Our approach is based on a standard global register allocator. The allocator is applied to a problem that implicitly encodes instruction scheduling decisions besides the problems of register assignment and spilling. The scheduling options are computed beforehand; after register allocation, the register assignments are mapped to a set of scheduling constraints, and the instructions are arranged accordingly. Using this scheme, instruction scheduling is guided directly by the register allocator's needs, based on the allocator's global model of spill costs.

The rest of this paper is structured as follows. Section 2 discusses register allocation, instruction scheduling, and their interactions. Section 3 introduces Register Reuse Scheduling (RRS), our novel approach for integrating register allocation with instruction scheduling decisions. Section 4 discusses our implementation of RRS and gives benchmark results, Section 5 presents related work, and Section 6 concludes.

2. REGISTER ALLOCATION AND INSTRUCTION SCHEDULING

Register allocation is the problem of assigning program values (or *virtual registers*) to the target architecture's physical registers. If all eligible registers are in use, other values must be *spilled*: Spilled values are saved to stack slots and must be reloaded before use. Spilling a value is costly because reloads are much more expensive than register accesses. Decisions of which values to spill use a measure of spill costs, which are based on the number and expected execution frequencies of necessary reload instructions (and often other factors as well). The aim of register allocation is to minimize spill costs. As a secondary objective, values connected by copy instructions may be *coalesced*, i. e., allocated to the same register, to eliminate the copies.

Formulations of the register allocation problem are based

on a notion of live ranges of values: A value is live at every program point between its definition and its last use. Values that might be eligible for assignment to the same register (or aliasing registers), but that have overlapping live ranges, are said to interfere. Interfering values may not be allocated to aliasing registers. The register allocation problem may be presented as an undirected interference graph with values as nodes and edges between interfering values. A register assignment may then be found by coloring the nodes with register names such that no two neighboring nodes are assigned the same color [Cha82, BCT94]. If the graph is not found to be colorable, heuristics select values to spill, remove those from the interference graph, and try another round of coloring.

For architectures with irregular register files, register allocation based on the partitioned Boolean quadratic problem (PBQP) has proved useful [SE02, HS06]. For our purposes, it suffices to view PBQP register allocation as a generalization of graph coloring: In PBQP, an edge in the problem graph does not simply denote an interference, but rather carries a cost matrix capable of expressing complex relationships. Infinite costs in the matrix can express constraints such as interferences between registers and their subregisters as well as architectural requirements such as register pairing. Negative costs can represent the benefits of coalescing. Like graph coloring, PBQP is an NP-complete problem, but efficient near-optimal heuristics are known for both.

Instruction scheduling concerns the arrangement of instructions within a basic block (a maximal single-entry, single-exit region of the program) to optimize some notion of program performance. Schedulers typically work on a representation of the block as a data dependence graph (DDG), a directed acyclic graph of instructions that captures dependences between instructions. Semantically valid schedules must respect all the dependences in the DDG. Uses of a (virtual or physical) register depend on the register's definition, and further definitions depend on uses of the previous one. If register allocation is performed before scheduling, unrelated values may be assigned aliasing registers. In such cases, additional dependences represented by DDG arcs may arise and restrict the freedom of the instruction scheduler.

The problem of instruction scheduling rose to prominence when pipelined processor architectures became widespread. In order to exploit the instruction-level parallelism provided by such processors, instructions must be ordered such that independent computations can proceed in parallel [GM86]. Overlapping independent computations means that unrelated instructions are scheduled between a value's definition and its uses. This lengthens the live ranges of all the values involved and increases the number of interferences. If aggressive scheduling before register allocation (*prepass* scheduling) is applied, the allocation problem thus becomes more difficult, and more spill code is needed.

Integrated approaches, discussed in more detail in Section 5, were introduced to alleviate the problems caused by the opposing goals of register allocation to minimize spills and instruction scheduling to maximize parallelism. These methods typically assume that the cost of a reload is not much larger than the costs of other relatively expensive instructions, and that a meaningful trade-off can be found between an acceptable level of spilling and good pipeline utilization. Most of these approaches are heuristic in nature and rely on using incomplete, approximate models of register pressure during scheduling.

The gap between processor clock speeds and memory speeds has widened considerably over the past decades. We believe that nowadays avoiding memory accesses due to spills is much more important than avoiding pipeline stalls of a few cycles. Especially for modern out-of-order processors, reduction of spills is more important than exposing instruction-level parallelism [VG99]. Our approach for integrated scheduling and register allocation, presented in the following section, is therefore aimed primarily at aggressive scheduling to minimize global spill costs; the fact that this reduces scheduling freedom is of less concern.

3. REGISTER REUSE SCHEDULING

In this section, we present Register Reuse Scheduling (RRS), an extension to register allocation that may also make instruction scheduling decisions if those can help avoid spills. Spills may be avoided by finding a physical register that the value may be allocated to. Registers may be reused between values whose live ranges do not overlap. The general idea behind RRS is therefore to make scheduling decisions that optimize register reuse, guided by the register allocator's global model of spill costs. For each pair of values that might be allocated to aliasing registers, we determine the DDG arcs needed to ensure that their live ranges do not overlap. If the overlap can be avoided in this way, we need not consider these ranges to interfere in the register allocation problem, since the values may then be allocated to aliasing physical registers. After allocation, if the values were indeed assigned to aliasing registers, we add the corresponding arcs to the DDG. Finally, we compute a schedule that respects all the register reuses chosen by the register allocator. An example demonstrating RRS is given in Section 3.4.

In contrast to many other integrated techniques for scheduling and register allocation (see Section 5 for a survey of related work), this approach does not attempt to use heuristics to *anticipate* the register allocator's needs during prepass scheduling; rather, implicit *sets* of valid schedules are presented to the allocator, and the allocator chooses those basic block schedules that help minimize its actual global spill costs. This results in a higher degree of integration of the two problems than approaches based on local scheduling heuristics.

3.1 Identification of register reuses

Traditional interference analysis before register allocation is based on fixed prepass schedules for basic blocks. Testing for overlap of live ranges is simple and results in one of two values: an overlap, meaning that the values interfere and may not be allocated to aliasing physical registers; or no overlap, meaning no interference.

In RRS, we perform a more involved three-valued analysis for values eligible for allocation to aliasing physical registers. In our case, we may find that values *definitely overlap*, meaning that they may not be allocated to aliasing registers; *do not overlap*, meaning no interference; or may present an

avoidable overlap, one that can be resolved by using an appropriate instruction schedule. We identify the kind of overlap by considering all the basic blocks in which both values are live at some point. If there are no such blocks, there is no overlap, while any block in which both values are live-in or live-out causes a definite overlap. Otherwise, within each of the blocks with possible overlaps, we search for possibilities to make the live ranges non-overlapping by considering additional DDG arcs: A reuse of aliasing registers for values v_i and v_j is possible if all uses of v_i can be forced to take place before all definitions of v_j, or vice versa. Such a reuse can be enforced by adding arcs to the DDG. This is legal if the dependence arcs from definitions to uses do not make the DDG cyclic; valid schedules are topological orderings of the DDG, so cyclic graphs cannot be scheduled.

Our analysis assumes that values may have several definitions, possibly in several basic blocks. (That is, we do not assume SSA form [CFR+91].) We therefore also accommodate for the possibility of DDG arcs, possibly in DDGs for more than one basic block, that force *interleaving* rather than sequencing of two values' live ranges. Such interleavings must ensure that only one of the values is ever live at one point.

When the register reuse analysis determines that a live range overlap between two values is possible but avoidable, it thus returns one or more sets of DDG arcs for each basic block in which the values may conflict. When viewed in isolation, each of these arc sets may be added to the appropriate DDGs without causing cycles (i. e., there will always be a valid schedule), and any schedule that respects the new DDGs will certainly not exhibit an overlap of the live ranges of the two values. The values may therefore be considered for allocation to aliasing registers.

3.2 Register allocation with reuse candidates

Based on the observation that spills may be avoided by facilitating register reuse, we want to maximize the register allocator's freedom for reuse. From the results of the reuse analysis described above, we select a maximally profitable set of *reuse candidates* to be presented to the allocator. In general, we cannot consider all the possible reuses identified by the analysis because, taken together, they would lead to cycles in the DDG. We therefore select a set of reuses of maximum expected profitability that does not cause DDG cycles (see Section 3.5). We use the values' spill costs to estimate expected profit. Values of greater cost would be more expensive to spill; as we want to minimize total spill costs, we attempt to facilitate reuse for values of greater spill cost.

Once a set of reuse candidates is chosen, we must also identify *reuse conflicts*. These are pairs of values identified as exhibiting a potential overlap, but for which no reuse was picked as a candidate (because it would introduce a cycle when added to the DDG and the candidate set selected so far). If the overlap is possible, but we did not select arcs to avoid it, we must be conservative and assume that the overlap will be present in the final schedule. In such cases, we may not allocate these two values to aliasing registers, so the register allocation problem must contain an interference for this pair of values.

We observe comparatively large candidate sets and small conflict sets in practice. This allows us to omit many possible interferences from the conflict graph. Only pairs of values that are definitively overlapping, and pairs from the relatively small conflict set, must be considered to interfere.

The allocation problem constructed using this method can be solved using an unmodified register allocator; the allocator need not be aware of the fact that its register assignments implicitly encode scheduling decisions. The set of reuses actually chosen by the allocator determines the level to which instruction schedules must be restricted. We believe that this approach is superior to register allocation based on any fixed prepass schedule, no matter how carefully it attempts to reduce interferences.

3.3 Scheduling

RRS starts with a set of DDGs for a function's basic blocks, computes reuse candidate sets, and builds a register allocation problem. The register allocator's solution encodes a set of arcs to be added to the DDGs to ensure legality of the allocation—i. e., adding those arcs ensures that the live ranges of values allocated to aliasing registers will not overlap.

For the scheduling step, we therefore consult the register assignments and the set of reuse candidates. For any pair of values that was a reuse candidate and that was indeed allocated to aliasing registers, we enforce legality of the reuse by adding the corresponding arcs to the DDG. In this manner, we obtain a restricted DDG that captures all the scheduling decisions implicitly made by the register allocator. Note again that these decisions are based on a global model of spill costs; further, as discussed above, two values may interact in more than one basic block. The global register allocator makes scheduling decisions that concern all the blocks in the function simultaneously, in contrast to schedulers that consider each block in isolation.

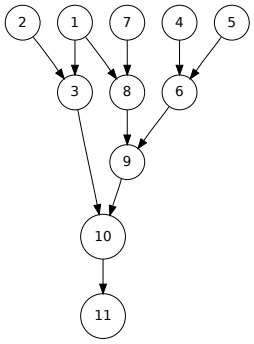
By design of RRS, the set of arcs associated with reuse candidates never causes DDG cycles, so there is always a legal schedule for the restricted DDG. We could apply any other additional arcs from the reuse set without causing an illegal DDG, but we want to restrict scheduling freedom only as much as is strictly needed to ensure the legality of the register allocation.

The remaining scheduling freedom in the restricted DDGs can be exploited by an unmodified aggressive scheduler without further regard for its impact on register allocation.

3.4 Example

We will demonstrate the ideas presented so far using a sample basic block adapted from Goodman and Hsu [GH88]. Figure 1 shows a basic block in an idealized machine language using virtual registers `Vi` on the left. Its data dependence graph shown in the center. In the DDG, we present dependence arcs as *precedence* arcs, i. e., an arc $a \to b$ means that instruction a must execute before b in any valid schedule. The virtual register interference graph for the given schedule is shown on the right of the figure. Virtual register `V10` does not interfere with any of the others because its live range starts after the end of all other live ranges.

```
 1  load   V1 ← a
 2  load   V2 ← b
 3  mul    V3 ← V1 * V2
 4  load   V4 ← c
 5  load   V5 ← d
 6  add    V6 ← V4 + V5
 7  load   V7 ← e
 8  add    V8 ← V1 + V7
 9  mul    V9 ← V6 * V8
10  add    V10 ← V3 + V9
11  store  h ← V10
```



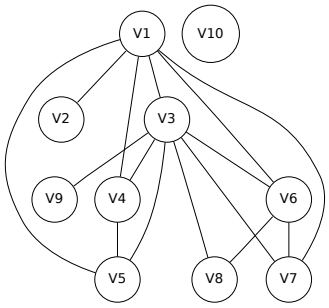


Figure 1: Example basic block, its data dependence graph (center) and the interference graph (right).

Table 1: Reuse candidates and conflict set.

Candidate	DDG arcs
V1 → V3	8 → 3
V4 → V1	6 → 1
V4 → V2	6 → 2
V4 → V3	6 → 3
V4 → V7	6 → 7
V4 → V8	6 → 8
V5 → V1	6 → 1
V5 → V2	6 → 2
V5 → V3	6 → 3
V5 → V7	6 → 7
V5 → V8	6 → 8
V6 → V2	9 → 2

Candidate	DDG arcs
V6 → V3	9 → 3
V7 → V2	8 → 2
V7 → V3	8 → 3
V8 → V2	9 → 2
V8 → V3	9 → 3

Conflict	DDG arcs
V1 → V8	3 → 8
V1 → V9	3 → 9
V2 → V9	3 → 9

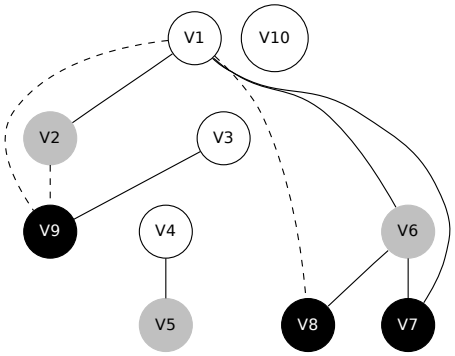


Figure 2: RRS interference graph for example block, with a valid 3-coloring.

Given the prepass schedule in Figure 1, spill-free register allocation requires at least four disjoint physical registers; the interference graph is 4-colorable but not 3-colorable. Assume that our target machine has only three registers available for allocation. We will show how RRS can compute a different schedule that minimizes spill costs (in this case, by avoiding all spills). This example, like our current implementation of RRS, presents it as an approach to *rescheduling*: changing a given schedule and register allocation problem according to the allocator's needs. However, a direct formulation of RRS on the DDG is possible, as described in the previous sections.

First, RRS identifies a set of register reuses and the sets of DDG arcs needed to guarantee the validity of each reuse by enforcing a relative ordering of instructions. As described above, the analysis determines one of three values for each pair of reuses: Definite overlap, definitely no overlap, or possible (avoidable) overlap. For instance, the live ranges for V1 and V2 definitely overlap since these values are both used by instruction 3. In contrast, live ranges for V4 and V9 cannot overlap: Existing DDG arcs already ensure that all uses of V4 (instruction 6) must be scheduled before all definitions of V9 (instruction 9). There are two (conflicting) ways to avoid the potential overlap between V5 and V7: Either instruction 8 must be forced to precede instruction 5, or 6 must precede 7.

Table 1 shows the results of reuse analysis partitioned into a large set of reuse candidates (left, continued top right) and other possible reuses that would conflict with the candidates by introducing cycles into the DDG (bottom right). A reuse denoted Vi → Vj means a reuse made possible by ensuring that the live range of Vi ends before the live range of Vj starts. It is easy to see that any of the arcs listed with the conflict set would cause a cycle when added to the DDG enhanced with the candidate arcs.

Once we have selected a set of register reuse candidates, we can relax the register allocation problem by removing any corresponding edges from the interference graph. For the example program, the candidates in Table 1 allow us to omit a set of eight edges from the interference graph in Figure 1, while the conflict set forces us to add the three interferences between V1 and V8, V1 and V9, and V2 and V9. We have thus reduced the overall number of interferences from 14 to 9, and we know that a schedule can be found that respects any valid register allocation on this relaxed problem. Figure 2 shows the relaxed interference graph of the example program. Removed reuse edges are omitted, while the new conflict edges are dashed. Note that the degree (number of neighbors) of most nodes could be reduced. In contrast to the original

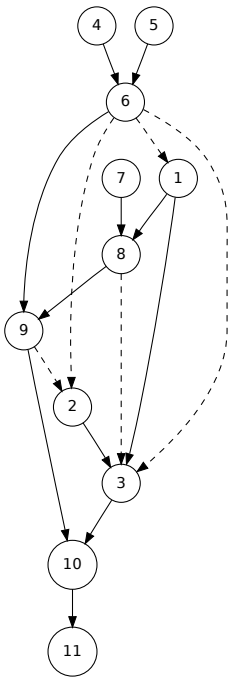


Figure 3: Example DDG with additional reuse arcs (dashed) as dictated by the reuse analysis of Table 1 and the register allocation of Figure 2.

graph, this relaxed interference graph does admit a coloring with only three physical registers. One possible coloring is indicated in the figure.

Relaxation of the register allocation problem allowed us to find a register mapping, but RRS must now ensure that a schedule is found in which live ranges allocated to the same physical register do not overlap. Using the mapping returned by the allocator, as shown by the coloring in Figure 2, we therefore consult Table 1 again and collect all the DDG arcs we need to ensure valid reuses. We can see that the following reuses were (implicitly) chosen by the register allocator: `V1` $\rightarrow$ `V3`, `V4` $\rightarrow$ `V1`, `V4` $\rightarrow$ `V3`, `V5` $\rightarrow$ `V2`, `V6` $\rightarrow$ `V2`. These reuses are associated with the arcs `8` $\rightarrow$ `3`, `6` $\rightarrow$ `1`, `6` $\rightarrow$ `3`, `6` $\rightarrow$ `2`, and `9` $\rightarrow$ `2`, respectively, all of which must now be added to the DDG. Figure 3 shows the DDG with these additional arcs in dashed style.

Note that, for example, the use of the same physical register for `V2`, `V5`, and `V6` was also legal in the prepass schedule, but we must now add DDG arcs to ensure that *any* schedule for the new DDG will respect the allocation of all of these values to the same physical register. Figure 4 shows one possible schedule for the example basic block after register allocation.

3.5 Dependence cycle elimination

In the discussion so far, we have mostly ignored the problem of selecting the DDG arcs that the register allocator may choose from. We must avoid selecting arcs that introduce

```
 4  load   R1 ← c
 5  load   R2 ← d
 7  load   R3 ← e
 6  add    R2 ← R1 + R2
 1  load   R1 ← a
 8  add    R3 ← R1 + R3
 9  mul    R3 ← R2 * R3
 2  load   R2 ← b
 3  mul    R1 ← R1 * R2
10  add    R1 ← R1 + R3
11  store  h ← R1
```

Figure 4: Example basic block after rescheduling according to Figure 3 and register allocation according to Figure 2.

cycles since cyclic DDGs cannot be scheduled. This means that register allocation may not select reuses which, taken together, would cause a cycle. We currently achieve this by partitioning the set of reuses into a candidate set that does not cause cycles, which we want to maximize, and a remaining conflict set. When building the register allocation problem, we avoid adding interferences for pairs of values in the reuse candidate set, but we must add interferences for pairs from the conflict set. This way, the register allocation problem is formulated based on an acyclic arc set, so no valid allocation can give rise to reuse arcs that cause DDG cycles.

Finding a minimum-cost conflict set is an instance of the (minimum) feedback arc set problem, which is known to be NP-complete [GJ79]. Rather than attempting an optimal solution, we select candidates based on a greedy heuristic: All potential reuses are sorted by descending total spill cost of the values involved. Starting with the highest-cost pairs, we collect an acyclic set of reuse candidates; any reuse that would cause a cycle is inserted into the conflict set.

Our experiments with selection of candidates based on this greedy heuristic gave encouraging results (Section 4). Conflict sets tend to be small, while candidate sets usually cover most of the potential reuses identified by the reuse analysis. In future work, we will evaluate the effects of a more disciplined but more expensive approach based on branch-and-bound. We also intend to research completely integrated formulations of the register allocation problem. These would deal directly with an input based on a cyclic set of reuse arcs, but would have to be guaranteed to return an acyclic result.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented a prototype of Register Reuse Scheduling within a development version of version 2.9 of the LLVM compiler framework[1], building on its existing PBQP register allocator. When constructing the PBQP problem, we examine each pair of virtual registers for potential register reuse and record the DDG arcs needed to enforce legality of the reuse.

Our implementation uses a dual strategy to minimize spill

[1] `http://www.llvm.org`

Table 2: Effects of Register Reuse Scheduling on static spill costs on SPEC CPU2000 benchmarks.

	PBQP		RRS		$\frac{\text{RRS}}{\text{PBQP}}$	
Benchmark	Spills	Spill Costs	Spills	Spill Costs	Spills	Spill Costs
164.gzip	168	3.83e+03	168	3.83e+03	1.000	1.000
175.vpr	1428	1.12e+04	1393	1.14e+04	0.975	0.999
181.mcf	42	3.99e+01	32	3.85e+01	0.762	0.966
186.crafty	2002	3.33e+02	1074	2.91e+02	0.536	0.875
197.parser	441	6.12e+03	437	5.18e+03	0.991	0.846
253.perlbmk	1445	1.23e+04	1539	1.23e+04	1.065	1.002
254.gap	2270	6.10e+03	2263	6.62e+03	0.997	1.086
255.vortex	1908	6.34e+03	1749	6.33e+03	0.917	0.998
256.bzip2	177	2.33e+03	176	2.32e+03	0.994	0.995
177.mesa	8821	1.94e+04	8779	1.91e+04	0.995	0.985
179.art	295	1.44e+04	257	1.23e+04	0.871	0.858
183.equake	1092	6.48e+02	1015	6.45e+03	0.929	0.995
188.ammp	7026	1.06e+04	6777	1.03e+04	0.965	0.979
geometric mean					0.911	0.966

costs: In each round of allocation, we solve both the original PBQP problem based on the prepass schedule and the problem relaxed for rescheduling. If RRS is able to find a solution that avoids spilling (possibly at the cost of rescheduling), that solution is chosen, all the required arcs are added to the DDGs, and the affected basic blocks are rescheduled. If RRS is not able to find a spill-free solution, neither will the original register allocator; however, we use the original allocator's choice of which values to spill. This gives better results than spilling based on the RRS problem because the RRS solution is tied to the assumption that certain reschedulings will be performed. However, we do not perform these reschedulings if we did not find a final, spill free allocation. We are investigating how to avoid this wasteful doubled effort in each round of allocation while preserving allocation quality.

Table 2 compares the numbers of values spilled and static spill costs of all spilled values using PBQP and RRS on a set of benchmarks from the SPEC[2] CPU2000 benchmark suite. (We used all the C and C++ benchmarks, but various issues with our current prototype prevented us from obtaining data for the 176.gcc, 252.eon, and 300.twolf benchmarks.) The numbers given refer to code generated for the ARM architecture. PBQP register allocation was performed on a prepass schedule for minimum register pressure. As we can see, LLVM's heuristic scheduler for minimum register pressure leaves room for improvement: On almost all benchmarks, RRS spills fewer values and incurs lower total spill costs than PBQP. On average (geometric mean), the number of spilled values is reduced by 8.9 %, and the static spill cost is reduced by 3.4 % by using RRS versus PBQP.

On the benchmarks where RRS incurs higher spill costs (253.perlbmk and 254.gap), the greedy heuristic algorithm to collect reuse candidates made bad choices. We expect these values to improve by using more sophisticated algorithms to choose candidates. Note that the reduction in spilled values does not have a simple correspondence with the reduction in spill costs: RRS aims at reducing spill costs, which model the expected number of reload instructions executed by the program. Depending on the program, an overall reduction of costs may be achieved by spilling a larger number of cheaper values, or fewer more expensive values. RRS may even lead to more spilled values, but still to lower costs, if those values cause comparatively few reloads.

[2] `http://www.spec.org/`

Table 3: Total compile times and code generation times on SPEC CPU2000 benchmarks.

Allocator	Total time (s)	Codegen time (s)
Linear scan	426	166
PBQP	654	395
RRS	2344	2083

As discussed in the introduction, we expect a lower number of reloads to result in an overall speedup. We believe that this holds for the general-purpose and high-end embedded processors that we target, while more refined tradeoffs would be necessary between spilling and scheduling for simpler embedded architectures.

Table 3 compares the compile-time costs of RRS across the entire benchmark set of Table 2 to two of LLVM's mature register allocators: Its implementation of an extended form of linear scan register allocator [SB07] and PBQP. Times, rounded to seconds, are shown for total compile time as well as for code generation only. As can be seen, both PBQP and RRS dominate code generation time. RRS scales considerably worse than the other allocators and would not be suitable for interactive use at the moment. This is due partly to the dual strategy outlined above that forces our current implementation of RRS to construct and solve about twice as many PBQP problem instances as the plain PBQP allocator; further, the live range overlap checks on the DDG needed for RRS are inherently more expensive than checks for a given prepass schedule. However, even this unoptimized research artifact does already scale to real-life programs well enough to be applicable to highly optimizing builds that are less time critical.

5. RELATED WORK

A handbook chapter by Govindarajan [Gov08] gives an overview of instruction scheduling, with a section on phase ordering and integrated approaches to instruction scheduling

and register allocation.

Similarly to RRS, Norris and Pollock [NP93] build the register allocation problem based on the DDG and attempt to eliminate interferences by adding DDG edges. However, their approach is heuristic and based on estimates to identify dependences that are expected to restrict the scheduler the least. Pinter [Pin93] introduced a related approach. Goodman and Hsu's DAG-driven register allocator [GH88] also uses the DDG, attempting to insert arcs that are expected to lengthen the schedule the least.

Goodman and Hsu's IPS [GH88] and Bradlee et al.'s more complex RASE [BEH91] are prepass schedulers that work with estimates of register pressure and attempt to schedule in a way that does not exceed a register use threshold.

Work by Berson et al. [BGS99], Touati [Tou01], and Xu and Tessier [XT07] considers variations of an integrated approach that builds on register reuse DAGs. These represent reuses that are valid under all possible schedules. Their measure-and-reduce approach identifies DAG regions with excessive register usage (i. e., possibly more simultaneously live registers than available). Such excessive usages are reduced by live range splitting or by inserting reuse arcs into the DAG and corresponding dependence arcs into the DDG.

Eriksson et al. [ESK08] discuss integrated code generation that considers instruction selection, instruction scheduling and register allocation as one problem. They can solve small problems optimally using integer linear programming; on larger instances, their genetic algorithm scales better and gives good results.

Govindarajan et al. [GYA+03] use near-optimal heuristics to schedule basic blocks for minimum register usage. It is not clear how their local approach compares to our approach guided by the global register allocator. We did not have the resources to implement alternative integrated scheduling and register allocation algorithms in order to compare them to RRS.

Ambrosch et al. [AEBK94] perform 'dependence-conscious' register allocation by coloring. Like ours, their interference graph is built based on the DDG. During coloring, the choice of register to assign to a value is guided by a cost metric. The metric is based on the scheduling impact of the additional dependences introduced by assigning certain registers. DDG and interference edges are added during the coloring process; this incremental approach avoids introduction of DDG cycles.

Our RRS algorithm can be contrasted with most of these works in one or more of the following points: First, our approach is aimed strictly at minimizing spill cost, not at a trade-off between spilling and instruction level parallelism. Second, our approach is global: Scheduling decisions are always guided by a global model of spill costs, not by considering basic blocks in isolation. Choosing the same register for a pair of values may also affect the schedules of several basic blocks simultaneously. Third, our spill costs are not estimates computed by the prepass scheduler, but rather the actual costs used by the register allocator.

Near-optimal instruction selection by Ebner et al. [EBS+08] is relevant to our work in the sense that they use PBQP to model a code generation problem and also need to avoid selection of cycles (cyclic instruction patterns). However, they can use a simple cycle-breaking scheme by relying on a topological ordering of patterns according to program order. This results in suboptimal selections on the local scale of individual instruction patterns. We cannot make such changes to a cyclic PBQP solution because they would have global effects on the register assignment.

6. CONCLUSIONS AND FUTURE WORK

This work introduced Register Reuse Scheduling (RRS), a new approach to integrated global register allocation and instruction scheduling. The aim of RRS is to find basic block schedules that allow maximal reuse of physical registers for independent values, and thus minimization of spill costs for the whole function. To this end, RRS allows the register allocator to make certain scheduling choices. We use a near-optimal PBQP register allocator with a relaxed problem based on data dependence graphs (DDGs) that allows more register reuses than fixed prepass schedules. Admissible reuses are associated with additional DDG arcs; the register allocator implicitly chooses among these arcs. Only the arcs needed to ensure the legality of the reuses chosen by the register allocator are added to the DDGs, which are finally scheduled using a postpass scheduler. We have shown that this method can reduce the static costs of the spill code generated by the PBQP register allocator by 3.4 % and the number of values spilled by 8.9 % on average on the SPEC CPU2000 benchmark suite.

In future work, we would like to explore further the potential of our approach. It appears that better selection of a set of reuse arcs may improve our results significantly; we intend to experiment with more powerful heuristics than the greedy approach taken in this work. We will also research the possibility of a fully integrated model in which the PBQP model or the solver itself ensures that no cyclic set of reuse arcs is ever chosen, even if presented with a cyclic set of reuse candidates as input.

Acknowledgments

This work was supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P21842, *Optimal Code Generation for Explicitly Parallel Processors*, `http://www.complang.tuwien.ac.at/epicopt/`.

Florian Brandner, Alexander Jordan, Andreas Krall, Viktor Pavlu, and the anonymous reviewers provided helpful feedback on earlier versions of this article.

7. REFERENCES

[AEBK94] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-conscious global register allocation. In *Proceedings of the International Conference on Programming Languages and System Architectures*, number 782 in Lecture Notes in Computer Science, pages 125–136, London, UK, 1994. Springer-Verlag.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16:428–455, May 1994.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 122–131, New York, NY, USA, 1991. ACM.

[BGS99] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Integrated instruction scheduling and register allocation techniques. In *LCPC '98: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 247–262, London, UK, 1999. Springer-Verlag.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[EBS+08] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA-graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 31–40, New York, NY, USA, 2008. ACM.

[ESK08] Mattias V. Eriksson, Oskar Skoog, and Christoph W. Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 11–20, New York, NY, USA, 2008. ACM.

[GH88] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GM86] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 11–16, New York, NY, USA, 1986. ACM.

[Gov08] R. Govindarajan. Instruction scheduling. In Y. N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2008.

[GYA+03] R. Govindarajan, Hongbo Yang, J. N. Amaral, Chihong Zhang, and G. R. Gao. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *Computers, IEEE Transactions on*, 52(1):4–20, Jan. 2003.

[HS06] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In David Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 346–361. Springer Berlin / Heidelberg, 2006.

[NP93] C. Norris and L. L. Pollock. A scheduler-sensitive global register allocator. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 804–813, New York, NY, USA, 1993. ACM.

[Pin93] Shlomit S. Pinter. Register allocation with instruction scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 248–257, New York, NY, USA, 1993. ACM.

[SB07] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *CC'07: Proceedings of the 16th international conference on Compiler construction*, pages 141–155, Berlin, Heidelberg, 2007. Springer-Verlag.

[SE02] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, LCTES/SCOPES '02, pages 139–148, New York, NY, USA, 2002. ACM.

[Tou01] Sid Ahmed Ali Touati. Register saturation in superscalar and VLIW codes. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 213–228, London, UK, 2001. Springer-Verlag.

[VG99] Madhavi Gopal Valluri and R. Govindarajan. Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 78, Washington, DC, USA, 1999. IEEE Computer Society.

[XT07] Weifeng Xu and Russell Tessier. Tetris: a new register pressure control technique for VLIW processors. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 113–122, New York, NY, USA, 2007. ACM.