Optimistic Integrated Instruction Scheduling and Register Allocation

Gergö Barany and Andreas Krall

Institute of Computer Languages, Vienna University of Technology
{gergo,andi}@complang.tuwien.ac.at

Abstract. Instruction scheduling and register allocation are two fundamental operations in an optimizing compiler's back-end. Scheduling is especially important in order to exploit parallel functional units in modern superscalar or VLIW architectures. There is a well-known phase ordering problem between these two stages: Performing either stage first can force the other stage to make suboptimal decisions.
We propose an *optimistic* integrated approach in which scheduling is performed before register allocation, but where not all scheduling decisions are final; rather, the register allocator may rearrange the order of certain instructions to find a good match of register usage to the actually available registers.
The rescheduling register allocator is based on a linear-scan allocator. The allocator passes over the program and assigns registers to live ranges. When at some point there are more live values than available machine registers, standard allocators make a decision to spill some value to memory or to split a live range. With our scheme, there is an additional possibility: Instructions may be rearranged so that some live ranges end earlier, freeing registers for other values. Such code motion may lead to longer schedules because more unfilled delay slots may be exposed. In many cases, we expect this to be preferable to more expensive spilling.
We evaluate a prototype implementation of our approach and find that it succeeds in eliminating some spills, but (as expected) not as many as a scheduler aimed at minimizing register use. To expose greater opportunities for aggressive optimizations, we will investigate our integrated approach in the setting of superblock scheduling for instruction-level parallel processor architectures.

1 Introduction

Among the passes in an optimizing compiler's back-end, instruction scheduling and register allocation are especially important ones. The goal of instruction scheduling is to find an ordering of machine instructions that minimizes execution time; register allocation is responsible for mapping program values to machine registers to minimize memory traffic.

The instruction scheduling phase is presented with a set of machine instructions from the instruction selection phase. The instructions may use virtual registers or physical registers as operands, depending on whether scheduling is performed before or after register allocation (prepass/postpass scheduling). There are dependences between some instructions: An instruction that produces a value must be scheduled before any instruction using that value (data dependence); all uses of a value must be scheduled before any instruction that overwrites the value (anti-dependence) [AK01].

These dependences typically do not induce a total ordering of the machine instructions. The instruction scheduler must choose from among the many valid schedules one that minimizes schedule length by the best possible utilization of processors' multiple functional units and pipelines. In particular, memory loads and other instructions that take many cycles to evaluate can be overlapped with independent instructions to fill their delay cycles.

Register allocation aims to allocate the most frequently used program values from virtual registers to machine registers and to minimize memory traffic by minimizing the number of stores (spills) and expensive reloads of values executed by the program. An important notion in register allocation is that of the live range: A value is live from its point of definition until its last use. Overlapping live ranges of different values cannot be allocated to the same physical register since the later definition would overwrite the earlier one and cause the wrong value to reach some of the uses. Conversely, non-overlapping live ranges may safely be allocated to the same physical register as there is no possibility of interference.

There is a well-known phase ordering problem between instruction scheduling and register allocation: Prepass scheduling that maximizes pipeline utilization may place many instructions between a virtual register's definition and its uses. This tends to lengthen live ranges and to increase the number of overlaps, often causing more spills. On the other hand, in postpass scheduling, allocation of different values to the same physical register can introduce false dependences that constrain the scheduler's choices. Thus, neither ordering of decoupled instruction scheduling and register allocation passes might result in an optimal program. Many cooperative or integrated approaches for solving the combined problem have therefore been proposed. This paper introduces a new technique based on rescheduling during register allocation, modifying the output of the prepass scheduler where necessary to avoid spilling.

The structure of the remainder of this work is as follows: The next section surveys related work in the field of combining scheduling and register allocation. Section 3 presents our approach of rescheduling register allocation and discusses various heuristics. Section 4 describes our implementation of a rescheduling register allocator and presents early experimental results. Section 5 discusses future work, and Section 6 concludes.

2 Related Work

Combined approaches to instruction scheduling and register allocation are a classic topic in compiler research. Most of them are cooperative, i. e., one of the phases attempts to take the needs of the following phase into consideration.

An early important contribution was integrated prepass scheduling (IPS) by Goodman and Hsu [GH88]. In IPS, a prepass scheduler is combined with a liveness analysis to estimate register pressure at the beginning of each basic block in the program. The register pressure model is adjusted during scheduling: Instructions that define a register increase register pressure, while the last use of a value frees the corresponding register. By default, the scheduler orders instructions to minimize exposed latencies. However, if register pressure rises above a predefined threshold, IPS switches to a mode that prefers to shorten live ranges. This approaches strikes a good balance between pipeline utilization and spill code reduction.

A more complex method called RASE [BEH91] relies on multiple scheduling passes: Each block is first tentatively scheduled twice, under the respective assumptions that register pressure is very high and very low. The tentative schedule lengths can be interpolated using a hyperbolic function to approximate the expected schedule length for any number of available registers. Using this information, a global analysis and register allocator can assign each block a 'register budget' so as to minimize the expected overall number of executed instructions. In the last pass, each block is assigned a final schedule and local register allocation that does not exceed the assigned register budget (introducing spill code if necessary). The authors found that the results produced by RASE were quite similar to IPS and did not warrant the increased complexity.

The URSA method of Berson et al. [BGS99] is based on register reuse DAGs which are used to identify groups of instructions that would use too many registers if scheduled in parallel. Edges in a register reuse DAG connect two instructions if the target instruction can reuse a register freed by the source; independent instructions, i. e., those that do not lie on a common path through the reuse DAG, cannot share registers and would increase register pressure if scheduled simultaneously. URSA uses live range splitting to reduce the number of interfering live ranges.

Touati's work on register saturation [Tou01] improves the heuristics used by URSA for determining the maximal register need for a DAG. Touati introduced an optimal integer linear programming formulation as well as near-optimal heuristics for computing and reducing this limit.

DAG-driven register allocation, introduced by Goodman and Hsu in the same paper as IPS [GH88], uses the data dependence graph for register allocation before a postpass scheduler. This approach exploits dependences already present in the graph: Where there is already a (possibly transitive) dependence from the end of one live range to the start of another, both can be allocated the same register without introducing false dependences.

Pinter [Pin93] extended the graph coloring approach commonly used for register allocation to include scheduling information. This approach is based on the complement of the graph representing (transitive) scheduling dependences. In this complement graph, an edge connects two instructions iff they can be executed in parallel. These edges are added to the register interference graph. Coloring the graph with physical registers such that no two adjacent nodes are assigned the same register will ensure that no false dependences are introduced between instructions. Norris and Pollock [NP93] presented a similar approach.

Ambrosch et al. [AEBK94] introduced dependence-conscious coloring. In this approach liveness analysis for register allocation is performed on the dependence graph, not on linear pieces of intermediate code. During register allocation, anti-dependence edges may be added to the graph due to some register assignments; registers are selected to minimize the number of non-redundant anti-dependences. This graph-based approach introduces fewer dependences than graph coloring register allocation on previously linearized code. This leaves more freedom to the postpass scheduler.

Various groups implemented code generators integrating optimal instruction selection, instruction scheduling and register allocation, based on formulations such as integer linear programming. We cite the work of Eriksson et al. as an example [ESK08].

Finally, on modern architectures with out-of-order execution, good pipeline utilization can be ensured dynamically by the CPU, which makes static scheduling less important. For such architectures, it is more important to avoid spills whenever possible, so scheduling aimed at minimal register pressure is an appropriate choice [VG99].

3 Rescheduling Register Allocation

We present a novel approach to the integration of instruction scheduling and register allocation. In this approach, the responsibility for balancing the needs of the two phases lies entirely with the register allocator. The prepass scheduler can be an aggressive off-the-shelf scheduler that aims only at reducing the schedule length without regard for register pressure. The register allocator may change the schedule during allocation according to its needs to avoid expensive spills. We expect this approach to be able to eliminate most of the scheduling decisions that turn out to hinder register allocation; it is this aspect of the framework that we refer to as *optimistic* scheduling, avoiding conservative register pressure estimates during scheduling.

3.1 Rescheduling During Linear Scan Register Allocation

Our rescheduling register allocator formulation extends linear scan register allocation [PS99]. Linear scan is a simple technique based on a serialization of the program's basic blocks, where a value's live range is approximated by a single interval of program positions that includes all uses and definitions of the value. (This simple approximation may include program points where the value is not even live.) The register allocator performs a single sweep over the program, allocating registers to each interval it reaches. The original linear scan approach was aimed at just-in-time compilation and traded compilation speed for somewhat worse code than produced by other allocators. Extended versions include various improvements and rival the code quality of graph coloring register allocators at much faster compilation speeds [SB07].

At the start of some live interval the linear scan allocator is not always able to assign a new physical register because all registers are used up. In such cases, it must usually make a decision to spill one of the overlapping values. It is at this point that our modified allocator offers another choice: Rather than introducing expensive spilling, it can attempt to reschedule some instructions to shorten some of the overlapping live ranges.

Moving a single instruction can shorten live ranges in the following way: The end of a live range (the last use of the corresponding value) may be hoisted to the point where the register allocator has run out of registers. The only instructions that can be hoisted are ones that can legally be scheduled at that point. In particular, this means that all of their operands must be live; in a linear scan allocator that builds upon the static single assignment form (SSA) [CFR$^+$91], this is the case iff all operands have been assigned to physical registers. Other conditions to ensure include correct ordering of possibly aliasing memory accesses, and correct handling of implicit condition code registers.

As described above, we only consider those instructions for hoisting that end live ranges. This means that after hoisting the instruction must still be the last use of a value; in other words, last uses may not be hoisted past other uses of the same value. In this case, the hoisted instruction will free the register(s) for which it is the last use.

The hoisted instruction may free more registers than it defines. This can be the case for instructions that do not define register values, such as stores to memory; it may also be the case for binary operations that are last uses of both of their operands simultaneously. If an instruction like this is hoisted, its register result (if any) can be allocated to one of the freed registers, and at least one other register is free for other use. Provided that register types and sizes match, this will typically allow the register allocator to successfully reconsider the instruction that blocked it previously. In such cases, a physical register can be assigned to that instruction's result, and spilling is avoided.

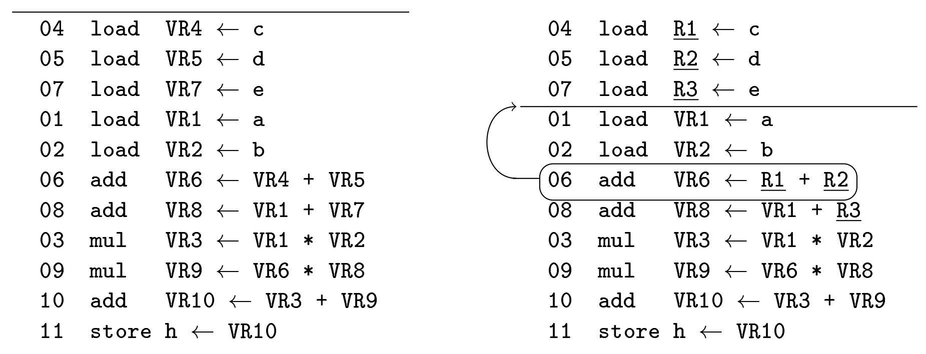
The more common case is that of a hoisted instruction that frees just as many registers as it defines, typically one. In this case, the register allocator can reuse the freed register for the instruction's result, but does not get any additional registers to assign to the following instructions. Some progress is still made, however: One more instruction can be assigned a register, and this rescheduling operation may enable further code motions that eventually free registers. Such repeated rescheduling operations can eventually avoid some spills as well, but it is impossible to tell this without performing lookahead operations.

As explained above, rescheduling may prevent the spilling of values to memory. This can be a very significant gain as memory accesses can incur long delays. The cost of rescheduling is a modified schedule in which the uses of certain values are closer to their definitions. This may cause pipeline delays if a defining instruction's latency is no longer fully masked by other intervening instructions. However, we expect this cost of a few cycles typically to be much less than a spill would cost. Unfortunately, heuristic rescheduling may shorten some live ranges at the expense of lengthening others, worsening the results of register allocation in other parts of the program.

Besides linear scan, a formulation of rescheduling register allocation based on graph coloring is also possible: Just as with linear scan, a value must be spilled if too many live ranges interfere at some point. Rescheduling to shorten some live range(s) removes interferences; this corresponds to cutting interference edges, which may make the graph easier to color. As with linear scan, this technique of removing interferences is orthogonal to spilling and may be considered as an alternative if deemed profitable. We will not follow this thread further and only consider linear scan register allocation for the rest of this paper.

3.2 Example

We will illustrate our approach using an example adapted from Goodman and Hsu [GH88]. Consider the basic block in Figure 1(a), scheduled to mask load latencies as well as possible. We will illustrate crucial steps in rescheduling linear scan register allocation on this basic block. The vertical line indicates the current position of the allocator; we assume that three physical registers are available for this basic block.



```
04  load   VR4 ← c
05  load   VR5 ← d
07  load   VR7 ← e
01  load   VR1 ← a
02  load   VR2 ← b
06  add    VR6 ← VR4 + VR5
08  add    VR8 ← VR1 + VR7
03  mul    VR3 ← VR1 * VR2
09  mul    VR9 ← VR6 * VR8
10  add    VR10 ← VR3 + VR9
11  store  h ← VR10
```

(a) Example basic block at the start of register allocation.

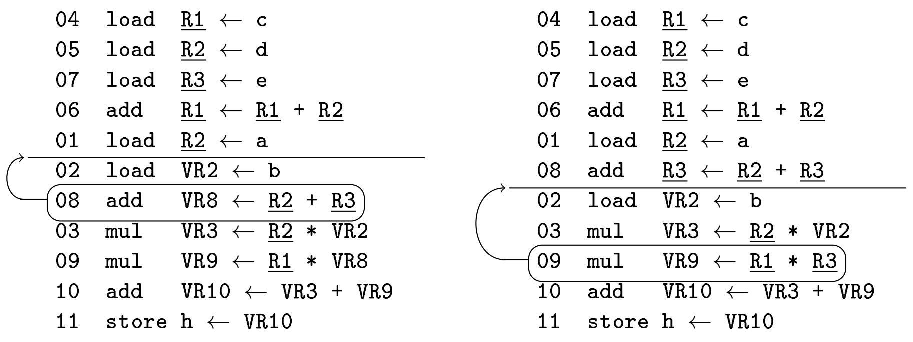
```
04  load   R1 ← c
05  load   R2 ← d
07  load   R3 ← e
01  load   VR1 ← a
02  load   VR2 ← b
06  add    VR6 ← R1 + R2
08  add    VR8 ← VR1 + R3
03  mul    VR3 ← VR1 * VR2
09  mul    VR9 ← VR6 * VR8
10  add    VR10 ← VR3 + VR9
11  store  h ← VR10
```

(b) Register allocator blocked after three steps; instruction 06 is available for rescheduling.

Fig. 1. Running example: Linear-scan register allocation with rescheduling, assuming three available physical registers.

The results of the first three instructions can be allocated to the three available physical registers as shown in Figure 1(b). (Physical registers are underlined for better visibility.) The register allocator is blocked after processing these three instructions as there are no more free registers to assign to the result of instruction 01. In this situation, the usual allocator would have to introduce spill code to free a register. The rescheduling allocator instead looks ahead to find instruction 06 ready for rescheduling as indicated in Figure 1(b). This instruction is available because all of its operands have already been assigned physical registers; in an allocator based on SSA form, this means that these registers will not be modified by intervening instructions, so their uses can be scheduled right away.

After hoisting instruction 06 before instruction 01, register allocation can continue because 06 is the last use of two registers; one of them is reused for the instruction's result, the other is free. Instruction 01 will now be considered again and can successfully be assigned the free register.



```
04  load  R1 ← c
05  load  R2 ← d
07  load  R3 ← e
06  add   R1 ← R1 + R2
01  load  R2 ← a
02  load  VR2 ← b
08  add   VR8 ← R2 + R3
03  mul   VR3 ← R2 * VR2
09  mul   VR9 ← R1 * VR8
10  add   VR10 ← VR3 + VR9
11  store h ← VR10
```

(a) Rescheduling freed registers for instructions 06 and 01 (blocked previously). Further rescheduling is needed because instruction 02 is blocked.

```
04  load  R1 ← c
05  load  R2 ← d
07  load  R3 ← e
06  add   R1 ← R1 + R2
01  load  R2 ← a
08  add   R3 ← R2 + R3
02  load  VR2 ← b
03  mul   VR3 ← R2 * VR2
09  mul   VR9 ← R1 * R3
10  add   VR10 ← VR3 + VR9
11  store h ← VR10
```

(b) Instruction 02 is still blocked after rescheduling, but progress has been made. Rescheduling instruction 09 will resolve all register conflicts.

Fig. 2. The example continued: After rescheduling, the register allocator is blocked again. Further rescheduling will finally allow successful allocation to three physical registers without spilling.

The example is continued in Figure 2. In Figure 2(a), the register allocator is now blocked at instruction 02, which is the last load operation. It must be delayed until a register can be freed. In Figure 2(a), instruction 08 can be hoisted, and a physical register reused for its result. This does not free a register for the previously blocked instruction yet. However, instruction 09 becomes available as indicated in Figure 2(b) because its second operand has now been bound to a physical register. After the last rescheduling operation, there are enough free registers to finally assign a register to instruction 02 and the rest of the basic block. The final result of rescheduling register allocation is shown in Figure 3.

The final basic block is scheduled such that its register need does not exceed the number of available registers. At the same time, some of the assumed load latencies are still masked by the schedule. An implementation if IPS could produce the same result in this example as the rescheduling allocator. Any final

```
04  load  R1 ← c
05  load  R2 ← d
07  load  R3 ← e
06  add   R1 ← R1 + R2
01  load  R2 ← a
08  add   R3 ← R2 + R3
09  mul   R3 ← R1 * R3
02  load  R1 ← b
03  mul   R1 ← R2 * R1
10  add   R1 ← R1 + R3
11  store h ← R1
```

Fig. 3. Final result of rescheduling register allocation applied to the running example. By rescheduling, the register limit could be met without spilling.

schedule derived by the rescheduling allocator is of course a valid schedule that could have been derived by a prepass scheduler. The advantage of integration into the register allocator is that decisions can be made on the basis of the actual register usage model of the allocator, not of pessimistic estimates beforehand.

3.3 Rescheduling Heuristics

In the description and the example given above, the only kind of code motion was the hoisting of instructions to earlier points. If such instructions can free or reuse registers, the previously blocked instruction may eventually be assigned a register without spilling. Rather than looking for profitable instructions to hoist, we might try a different approach: The blocked instruction could be delayed as far as possible, i. e., until just before the first use of its result, or until the end of the current basic block.

Our approach currently chooses the closest instruction that is legal and profitable to hoist. This is the same instruction that would be the next to be processed by the register allocator if one or more instructions had to be delayed. Thus, the local effect of hoisting and delaying appears to be the same. The difference is in the position where the previously blocked instruction will end up: When we use hoisting, it will be delayed as little as possible; sinking it down to the latest possible position will perturb the schedule more, but it may lower register pressure over a longer stretch of code.

In the future, we would like to investigate both experimentally and in a more formal way the possible rescheduling heuristics and their relationship to established techniques such as IPS.

4 Implementation and Evaluation

We implemented a prototype of the rescheduling register allocator described above using the LLVM compiler framework [LA04]. This section describes some implementation details and gives a detailed experimental evaluation of our prototype.

4.1 Implementation in LLVM

The LLVM compiler framework is a great tool for exploring optimization and code generation techniques. LLVM comes with back-ends for various architectures and provides a number of target-independent optimizations, instruction selectors, schedulers and register allocators. To validate our concept of rescheduling during register allocation, we modified LLVM's default register allocator, which is based on linear scan. This allocator includes some backtracking to try an improved allocation when it spills a value, so it is not actually 'linear' anymore; this backtracking is orthogonal to our approach and does not require any special handling.

When the register allocator fails to assign a physical register, it usually invokes code to spill a value instead. At this point, we added a check for reschedulability: We need only examine the list of currently 'active' live ranges. (This list's length is bounded by the number of physical registers.) We identify those active live ranges that are of a correct register class, and which end in the basic block in which the register allocator is currently blocked. Of these candidates, we prefer the closest instruction that ends a live range (to disturb the schedule as little as possible) which is legal to reschedule: In a simple linear pass over all intervening instructions, we check that we will not disturb the relative ordering of memory writes and reads, that no condition code register values will be clobbered by the rescheduling, and that there are no further uses of the live range we want to end.

After having chosen a legal candidate, it is moved in the list of instructions in its basic block. A linked list containing liveness information must also be updated in a simple linear pass to ensure that the analysis information is in sync with the modified code. All modifications of the program and analysis data are strictly local. If rescheduling fails due to a lack of legal candidates, we fall back on LLVM's regular spilling mechanism.

This approach appears easy to extend to superblock scheduling. For this, we need only relax the condition that the ends of candidate live ranges be in the same block as the current position of the register allocator. The legality checks must make sure to examine the effects of all possibly intervening blocks.

Finally, we implemented not only the hoisting heuristic, but also the sinking heuristic described in Section 3.3. The results for the two heuristics were virtually identical; the data given below were collected with the hoisting rescheduler.

4.2 Experimental Evaluation

We used the freely available MiBench[1] benchmark suite [GRE+01] to evaluate the prototype implementation of our rescheduling register allocator. Out of a

[1] http://www.eecs.umich.edu/mibench/

total of 24 programs in the suite, we chose the 15 programs that were not miscompiled by the development version of LLVM that we used as the basis of our prototype.

The tests were run on an Intel Xeon CPU running at 3 GHz under Linux. We do not currently have a production VLIW back-end to test our algorithm's effects on explicitly parallel processors.

For our tests, we used LLVM's latency-oriented top-down list scheduler (TD) as the baseline, and its bottom-up register pressure reducing scheduler (BURR) as an estimation of the maximum potential of eliminating spills. The rescheduling allocator builds on TD as its prepass scheduler and modifies its results where needed. We determined static spill and reload counts as well as counts of dynamic memory accesses (total reads and writes; these figures include *all* memory accesses, not only spills and reloads).

Table 1. Static results of the experimental evaluation of the rescheduling register allocator (Resched) compared to top-down latency-oriented scheduling (TD) and bottom-up register pressure reducing scheduling (BURR).

	Static spill sites			Static reload sites		
Benchmark	TD	Resched	BURR	TD	Resched	BURR
security-blowfish	33	18	12	49	34	25
security-rijndael	7	7	7	12	12	12
consumer-lame	600	602	526	802	803	708
consumer-mad	614	597	461	839	833	598
consumer-typeset	730	726	717	2810	2799	2808
consumer-jpeg	190	189	164	254	253	237
telecomm-adpcm	0	0	0	0	0	0
telecomm-fft	19	18	16	31	29	27
telecomm-gsm	14	14	9	28	28	12
automotive-bitcount	6	6	6	3	3	3
automotive-qsort	2	2	2	2	2	2
automotive-susan	709	702	641	950	945	876
network-patricia	1	1	1	1	1	1
network-dijkstra	0	0	0	0	0	0
office-stringsearch	0	0	0	0	0	0
Total	2925	2882	2562	5781	5742	5309
Percentage of TD		98.5 %	87.6 %		99.3 %	91.8 %

Table 1 summarizes the static data we obtained. We expected the rescheduling allocator to save some spills (where possible) over the latency-oriented scheduler, but to avoid fewer spills than the register pressure oriented scheduler. This is the case for all of the benchmarks except consumer-lame, where the rescheduling heuristic makes a bad decision that causes a slight increase in the number of spills. Interestingly, for the consumer-typeset benchmark, rescheduling spills more values than the register pressure oriented scheduler; however, due to a more fortunate choice of values to spill, this results in fewer static reload sites.

Table 2. Dynamic results of the experimental evaluation of the rescheduling register allocator (Resched) compared to top-down latency-oriented scheduling (TD) and bottom-up register pressure reducing scheduling (BURR). Figures for each benchmark are rounded to millions; summary data were computed from the unrounded numbers.

	Dynamic stores (mil.)			Dynamic loads (mil.)		
Benchmark	TD	Resched	BURR	TD	Resched	BURR
security-blowfish	78.8	75.6	75.6	227.6	220.8	220.8
security-rijndael	24.4	24.4	24.4	132.1	132.1	132.1
consumer-lame	84.4	84.4	86.3	341.0	341.0	333.1
consumer-mad	45.2	44.8	42.5	132.1	131.5	119.1
consumer-typeset	105.9	105.9	105.9	206.4	206.4	206.4
consumer-jpeg	8.7	8.7	8.8	26.8	26.5	26.7
telecomm-adpcm	6.8	6.8	6.8	40.1	40.1	40.1
telecomm-fft	39.1	39.1	38.8	67.7	67.7	67.4
telecomm-gsm	32.8	32.8	32.5	288.2	288.2	287.9
automotive-bitcount	27.0	27.0	27.0	68.7	68.7	68.7
automotive-qsort	40.8	40.8	40.8	73.5	73.5	73.5
automotive-susan	0.3	0.3	0.3	76.1	76.1	76.1
network-patricia	66.2	66.2	66.2	113.1	113.1	113.1
network-dijkstra	12.0	12.0	12.0	109.8	109.8	109.8
office-stringsearch	0.8	0.8	0.8	0.9	0.9	0.9
Totals	573.0	569.3	568.5	1904.0	1896.3	1875.8
Percentage of TD		99.4 %	99.2 %		99.6 %	98.5 %

Table 2 presents the dynamic memory access counts we obtained using the Cachegrind tool [Net04], part of the Valgrind dynamic analysis framework[2]. As expected, some correlation with the static data can be observed: Fewer static spills and reloads typically result in fewer dynamic spills and reloads as well. There are some anomalies, in particular with the consumer-jpeg benchmark: While the register pressure reducing scheduler (BURR) eliminates a larger number of static spills and reloads, it causes *more* dynamic loads and stores than the rescheduling register allocator. Such effects are probably due to unfortunate heuristic choices. In other cases, such as security-blowfish, some of the removed spills appear to be executed very infrequently: Although BURR eliminates somewhat more static spills and reloads than the rescheduler, the dynamic results are essentially identical.

Overall, the experimental data confirm the expected result that rescheduling during register allocation can succeed in reducing register pressure and thus eliminate some expensive spills. This approach does not avoid as many spills as a scheduler aiming for minimal register pressure, but it may be a workable trade-off between schedule length and register usage.

Figure 4 visualizes how much of the potential to reduce spills and reloads is realized by our approach. We again use TD as the baseline and use BURR to approximate the best possible reduction in spills and reloads. In the graph,

[2] http://valgrind.org

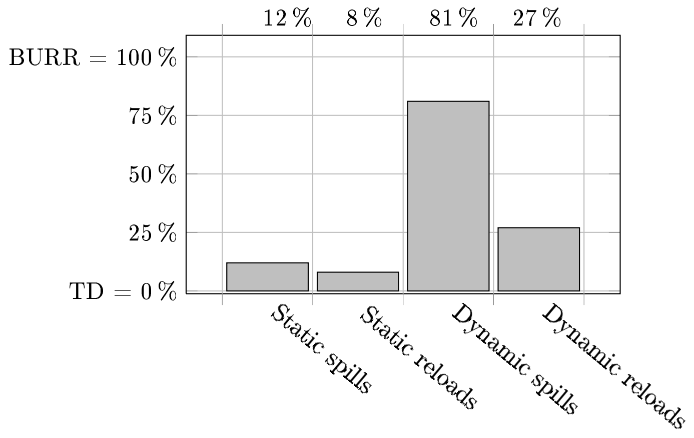


Fig. 4. Graph of the relationship between rescheduling register allocation and BURR. Each bar represents how much of the reduction potential is realized by our allocator. Higher bars denote more eliminated spills or reloads.

each bar corresponds to one of the measures in Tables 1 and 2. A value of 0 % corresponds to behavior like TD (no additional loads/stores eliminated), a value of 100 % would mean full realization of the potential to reduce spill code. The results show that our allocator's static effects appear small. However, it manages to eliminate 81 % of all dynamically executed spills that we may realistically hope to eliminate. The potential to reduce dynamic reloads is realized to 27 %. We hope to be able to improve this value by implementing a cost model for choosing between several candidates for code motion.

We would have liked to compare our integrated approach to established approaches; unfortunately, our prototype implementation of IPS produces too low quality code to allow a fair comparison.

5 Future Work

The work reported in this paper is at a somewhat early stage; the results in Section 4 are mostly a proof of concept that the general approach of rescheduling register allocation can make profitable modifications to avoid spilling after an aggressive prepass scheduler. While this work in progress could not yet be compared to other fast combined heuristic methods, we are encouraged to continue exploring the design space of rescheduling register allocation. Here, we sketch three main areas for future work in this field:

First, the rescheduling heuristics we implemented so far only account for a certain class of code motions; more general transformations, possibly moving more than one instruction at a time, might find more profitable schedules and register mappings. A general formulation should also include an estimation of the costs and advantages of code motions compared to the register allocator's model of spill costs.

Second, the scope of rescheduling could be extended from basic blocks to superblocks: We can move code between sets of basic blocks controlled by the same set of conditions, for example the block before a branch and after the corresponding join. Superblock scheduling is especially important to exploit the resources provided by explicitly parallel architectures such as VLIWs, and while it is a difficult problem in general, a superblock extension of rescheduling register allocation might be quite simple and yet profitable.

Finally, we intend to investigate rescheduling extensions of optimal or near-optimal register allocators based on formulations such as integer linear programming or partitioned Boolean quadratic problems (PBQP). In particular, LLVM's near-optimal PBQP register allocator does not include spilling in its formulation; if it cannot solve a register allocation problem, it requests an external component to spill some value before trying again. At this point, we might choose to reschedule instead of spilling. We would like to do this in a way that preserves as much of the solver's state as possible, which could make the restart of the PBQP register allocation considerably more efficient.

6 Conclusions

We presented an optimistic integrated scheme for instruction scheduling and register allocation: Unlike many other approaches that attempt to estimate and control register pressure during prepass scheduling, we use an aggressive standard scheduler. Register pressure is reduced by rescheduling instructions during register allocation, when precise data about register pressure and the allocator's model of spill costs are available. We believe that this may result in better balanced trade-offs than methods based on cooperative prepass schedulers.

The experimental results of our prototype implementation using the LLVM compiler framework encourage us to continue work in this area. We plan to investigate various rescheduling heuristics, extensions to superblock scheduling, and integration with optimal or near-optimal register allocators.

Acknowledgements

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/.

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.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Elsevier, 2001.

[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.

[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.

[GRE$^+$01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[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.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, September 1999.

[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.

[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.