SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2297

Experimental evaluation of various register-pressure-reduction heuristics

Ghassan Shobaki*,†, Laith Sakka, Najm Eldeen Abu Rmaileh and Hasan Al-Hamash

Princess Sumaya University for Technology, Al-Jubaiha, Amman, Jordan

SUMMARY

Minimizing the amount of spill code is still an open problem in code generation and optimization. The amount of spill code depends on both the register allocation algorithm and the pre-allocation instruction scheduling algorithm that controls the register pressure. In this paper, we focus on the impact of pre-allocation instruction scheduling on the amount of spill code. Many heuristic techniques have been proposed to do instruction scheduling with the objective of minimizing register pressure and consequently the amount of spill code. However, the performance of these heuristic techniques has not been studied relative to optimality on real large-scale programs. In this paper, we present an experimental study that evaluates the performance of several pre-allocation scheduling heuristics. The evaluation involves computing an experimental lower bound on the size of gap between each heuristic's performance and optimal performance. We also propose a simple heuristic technique based on a specific permutation of two basic priority schemes and experimentally evaluate the performance of this technique compared with other heuristics, including the heuristics implemented in the LLVM open-source Compiler. The evaluation is carried out by running SPEC CPU2006 on real x86-64 hardware and measuring both the amount of spill code and the execution time. The results of our study show that the proposed heuristic technique gives better overall performance than LLVM's best heuristic on x86-64, although it produces slightly more spilling. The proposed heuristic has better overall performance, because it achieves a better balance between register pressure and instruction-level parallelism (ILP). This result shows the importance of ILP in pre-allocation scheduling even on out-of-order machines. Furthermore, the results of the study show that there is a large gap between the performance of any of the studied heuristics and optimal performance; even the best heuristic in the study produces significantly more spill code than the optimal amount. This experimental result quantifies the intuitive belief that it is unlikely to find a heuristic that works well in all cases, thus showing the need for more rigorous solutions using combinatorial approaches. The paper discusses the challenges and complexities that are involved in developing such rigorous solutions. Copyright © 2014 John Wiley & Sons, Ltd.

Received 28 February 2014; Revised 18 August 2014; Accepted 29 September 2014

KEY WORDS: compiler optimizations; instruction scheduling; register pressure reduction; spill code minimization; performance optimization; combinatorial optimization; NP-complete problems; experimental algorithms

1. INTRODUCTION

Register allocation is a compiler optimization in which the objective is allocating as many *virtual registers* (program variables or compiler generated temporary variables) as possible to CPU registers (physical registers). A virtual register that cannot be allocated to a CPU register is *spilled* to main memory by adding a load before every use and a store after every definition of the virtual register. These loads and stores are referred to as *spill code*. The amount of spill code depends on both

*Correspondence to: Ghassan Shobaki, Computer Science, Princess Sumaya University for Technology, Al-Jubaiha, Amman, Jordan.
†E-mail: g.shobaki@psut.edu.jo

the register allocation algorithm and the instruction scheduling algorithm that is run before register allocation (*pre-allocation instruction scheduling*). Pre-allocation scheduling controls the *register pressure* (RP), which is the number of virtual registers that are simultaneously live and thus must be assigned to different physical registers. Therefore, presenting the register allocator with an instruction sequence that has a higher register pressure is likely to lead to more spill code. In this work, we study the role played by pre-allocation instruction scheduling in minimizing spill code by finding an instruction order that reduces the RP. The experimental results of this paper show that pre-allocation scheduling has a significant impact on the amount of spill code generated by the register allocator.

The pre-allocation instruction scheduling problem is further complicated by the fact that minimizing register pressure is not the only objective of the instruction scheduler. The scheduler has another important objective, which is exploiting instruction-level parallelism (ILP) by overlapping the execution of independent instructions. The two objectives of minimizing RP and overlapping independent instructions inherently conflict with each other. Overlapping more independent instructions naturally increases the overlap among virtual registers, thus increasing the RP. Therefore, a pre-allocation scheduler has to balance these two conflicting objectives. Many modern machines, however, have out-of-order execution engines that exploit ILP at run time. In some recent previous work [1, 2], it is assumed that when the target processor has out-of-order execution, the compiler scheduler can safely ignore ILP and schedule for the sole objective of minimizing RP. The experimental results of this paper show that this assumption is not always true and that compiler scheduling for ILP can in some cases have a significant impact on performance on out-of-order machines. The results show, however, that the best overall performance on out-of-order machines is achieved if RP is treated as a primary objective and ILP is accounted for as a secondary objective.

Instruction scheduling with the objective of minimizing register pressure and consequently the amount of spill code is known to be NP-hard [1]. Therefore, production compilers use heuristic techniques to solve this problem. Many heuristic techniques have been proposed for performing pre-allocation instruction scheduling, and it is widely accepted among researchers and practitioners that no heuristic gives optimal solutions in all cases. However, this common belief has not been quantified; that is, no experimental study has been performed to estimate the size of the gap between the performance of common heuristics and optimal performance on real programs. In this paper, we present an experimental study that quantifies this common belief. The heuristic techniques included in the study are the different combinations and permutations of two basic priority schemes (critical path and last use of a live register) that have been mentioned in the literature [3] as well as the set of heuristics implemented in the LLVM Compiler. The study also includes the combinatorial scheduling algorithm that has been previously proposed by the authors [4]. The evaluation is carried out using SPEC CPU2006 on real x86-64 hardware. The results of this experimental study show the following:

1. The size of the gap between the performance of the studied heuristics and optimal performance is fairly large. The best heuristic in the study produces significantly more spill code than the optimal amount of spill code. This quantification of the common belief shows the difficulty of the problem and the need for more rigorous solutions. The paper discusses the challenges and complexities that are involved in developing such rigorous solutions. This discussion explains why the difficulty of the problem goes well beyond the NP-completeness of any combinatorial formulation.
2. Compiler scheduling for ILP can in some cases have a significant impact on overall performance, even on modern out-of-order processors. Therefore, the objective of exploiting ILP cannot be totally ignored in pre-allocation scheduling.
3. One specific permutation of the two basic priority schemes (critical path and last use of a live register) outperforms LLVM's best heuristic on x86-64 in overall performance, while all other combinations and permutations of these priority schemes give significantly poorer performance. To the authors' best knowledge, no previous experimental study has been conducted to evaluate the performance of all combinations and permutations of these basic priority schemes. Therefore, this experimental finding of the paper is believed to be quite valuable

DOI: 10.1002/spe

from a practical point of view, because of the simplicity of these priority schemes compared with previously proposed heuristics [5, 1, 2]. Until a fundamentally superior algorithm is developed for solving this problem, we propose using this permutation as a simple but effective heuristic for doing pre-allocation instruction scheduling on out-of-order machines. The proposed heuristic is fairly easy to implement and gives good performance in a straightforward manner without any need for tuning or special case handling. The experimental results of this paper suggest that investing time and effort in developing a complex heuristic for pre-allocation scheduling is unlikely to be worthwhile in practice.

The rest of the paper is organized as follows. Section 2 defines the problem. Section 3 summarizes related previous work. Section 4 describes the proposed heuristic. Section 5 discusses the complexities that are involved in developing an optimal solution to the problem. Section 6 presents the experimental evaluation, and Section 7 concludes with a summary of our findings and briefly discusses possible future work.

2. PROBLEM DEFINITION

The problem addressed in this paper is pre-allocation instruction scheduling with the objective of achieving the best possible balance between minimizing spill code and exploiting ILP. Minimizing spill code is achieved by minimizing register pressure. The register pressure at a given point in an instruction sequence is the number of overlapping live registers (live ranges) at that point. As mentioned earlier, the two objectives of minimizing RP and exploiting ILP are inherently conflicting, because exploiting ILP requires overlapping the execution of as many independent instructions as possible, which leads to more overlap among live ranges, thus increasing RP.

In this paper, we consider heuristic approaches to solving this problem. Like many heuristics in compiler optimizations, the heuristics considered in this paper do not have explicit objective functions.[‡] Therefore, no formal definition of the scheduling objective is given. Rather, the performance of multiple heuristics is evaluated experimentally using two primary metrics: the amount of spill code generated and the actual execution speed.

The scope of the paper is limited to instruction scheduling within a basic block (*local instruction scheduling*). Because of the complexities of global instruction scheduling, which moves instructions across basic blocks [6], many production compilers, including LLVM, only implement local instruction scheduling. The input to a local instruction scheduler is a data dependence graph (DDG) representing dependencies among instructions. In pre-allocation scheduling, instruction operands are *virtual registers* that have not yet been assigned to physical registers.[§] The input to the pre-allocation scheduler must include each instruction's Def and Use sets. The Def set of an instruction is the set of virtual registers that are defined (produced) by the instruction, and the Use set is the set of virtual registers that are used (consumed) by the instruction. Given the Def and Use sets, the instruction scheduler can construct in a preprocessing step a list of virtual registers and store in the data structure that represents each virtual register a list of instructions that define this register and a list of instructions that use it.

An instruction schedule is an assignment of instructions to machine cycles. The *schedule length* is the estimated number of cycles needed to execute the instruction stream. The number of cycles is estimated based on a certain *machine model*. The machine model is a description of the target processor, including functional units, instruction latencies, and a mapping between instructions and functional units. Our implementation of the studied heuristics supports a general machine model with arbitrary functional units and latencies. The experimental results, however, are based on LLVM's simple machine model for x86, which assumes a single-issue machine. Our experience with machine modeling for x86 has shown that, because of the complexity of the x86 architecture and the massive out-of-order execution, it is hard to develop a compiler machine model for ILP

[‡]The experimental evaluation includes a combinatorial scheduler that has an explicit objective function.
[§]In certain special cases, the compiler may assign some virtual registers to physical registers before pre-allocation scheduling.

DOI: 10.1002/spe

scheduling on x86. Therefore, a simple machine model may give comparable performance to that achieved with a complex machine model.

3. PRIOR WORK

The problem of reducing register pressure and balancing that with ILP during pre-allocation scheduling has been studied by many researchers. Heuristic solutions to this problem have been proposed by Goodman and Hsu, Berson *et al*., Touati, Govindarajan *et al*. and Barany and Krall [5, 7, 8, 1, 2]. Combinatorial solutions have been proposed by Kessler, Govindarajan *et al*., Malik, Barany and Krall and Shobaki *et al*. [9, 1, 10, 2, 4]. In this section, we cover a selected subset of the heuristic and combinatorial techniques that have been proposed for solving this problem, focusing on the most recent attempts.

Berson *et al*. [7] propose a unified formulation of scheduling and allocation as one resource allocation problem by treating both registers and functional units as resources. The technique constructs a directed acyclic graph (DAG) representing resource requirements and identifies the regions that have excessive resource requirements (high degrees of parallelism or high register pressures). Certain heuristics are then used to reduce these excessive requirements by adding DAG edges that sequentialize some groups of instructions.

Govindarajan *et al*. [1] present both a heuristic technique and a combinatorial technique for minimizing RP alone, ignoring ILP. Their heuristic is based on forming instruction *lineages* in the DDG. A lineage is a set of instructions that can reuse the same destination register. Then, they construct a lineage interference graph (LIG) that captures unavoidable overlaps among lineages. The LIG is then colored using a graph-coloring heuristic. Finally, an instruction sequence is generated using a modified form of list scheduling. Clearly, the performance of this multistage algorithm depends on the heuristics that are used in each stage: lineage formation, graph coloring, and sequence generation. To evaluate their heuristic technique, Govindarajan *et al*. implemented a combinatorial algorithm using an integer linear programming formulation. Their combinatorial algorithm, however, was only applied to 675 small DDGs with an arithmetic-mean size of 19 instructions per DDG, because the algorithm timed out on larger DDGs. Comparing their heuristic solutions to the exact solutions, they found that their heuristic produces the optimal solution to 96% of the 675 DDGs. This experiment, however, is not sufficient to judge the performance of the heuristic, because of the small sizes of the DDGs used in the experiment. Past experience has shown that such small DDGs are usually easy to solve heuristically. For example, Shobaki *et al*. [4] report that the average size of the DDGs in FP2006 that could not be scheduled optimally for x86-64 using the CP heuristic is 99 instructions. The challenge then lies in optimally scheduling basic blocks with hundreds of instructions. Unfortunately, these larger blocks tend to have higher impacts on the actual execution times. The algorithm of Govindarajan *et al*. also ignores ILP, while our experimental evaluation shows that compiler scheduling for ILP can still have a significant impact on the performance of compute-intensive programs.

Kessler [9] presents the first combinatorial algorithm for solving the register pressure minimization problem in pre-allocation scheduling. Kessler's algorithm is based on a combination of branch-and-bound enumeration and dynamic programming. He describes both a pure RP minimization algorithm and an ILP-aware RP minimization algorithm. Kessler reports that the best form of his pure RP algorithm runs out of space and time on nearly all DDGs with more than 50 instructions and that his ILP-aware algorithm runs out of space for DDGs with more than 25 instructions. These results show that it is quite challenging to solve this problem optimally using a combinatorial approach. A few researchers later on worked on this problem using a combinatorial approach.

Malik [10] presents a combinatorial algorithm using constraint programming for solving the problem of basic block scheduling without spilling; that is, finding a minimum-length no-spill schedule if such a schedule exists. This problem has limited practical value, because, as shown in the experimental results of the current paper, most floating-point (FP) benchmarks have spill code in their hot functions. Malik's algorithm solves DDGs with up to 50 instructions within 10 min per DDG.

Barany and Krall [2] present a technique that reduces register requirement by reordering instructions in the register allocation phase. The technique identifies live ranges that may reuse the same register and adds edges to the DDG to prevent these live ranges from overlapping, thus reducing RP. Barany and Krall's technique has the advantage of using the *global* spill costs computed by

DOI: 10.1002/spe

the register allocator, as opposed to using pre-allocation estimates of register requirements. That leads to a stronger correlation between the scheduling objective and the spill cost, but it is not as accurate as formulating scheduling and allocation as a unified combinatorial optimization problem with a single objective. Barany and Krall [2] describe both a heuristic and a combinatorial algorithm (using integer linear programming) for selecting the live ranges that can reuse the same register. Their algorithm also has the advantage of allowing global code motion across basic blocks. However, both algorithms (combinatorial and heuristic) are pure RP-reduction algorithms that ignore ILP. As mentioned earlier, ILP is an important factor in some compute-intensive programs.

Barany and Krall [2] report large increases in compile times when their combinatorial algorithm is used. Yet, the improvements in execution times are relatively small. They report a total compile time of 18 h for 20 SPEC CPU2000 benchmarks and a geometric-mean improvement of 0.5% in execution time relative to LLVM's heuristic. The maximum improvement on a single benchmark is only 3%. These modest improvements in execution times are attributed to the exclusion of large functions with more than 1000 instructions (3% of the functions) and the timeouts on 4% of the functions included in the experiment. Although these percentages may appear small, our experimentation has shown that hot functions that have higher impacts on the performance of compute-intensive programs tend to be large and hard to schedule because of the high degrees of parallelism that they generally have. Barany and Krall's heuristic solution to the problem does not give an overall improvement relative to LLVM's heuristic. It degrades the performance of many benchmarks, with a geometric-mean degradation of 1%. This is another piece of experimental evidence showing the difficulty of solving this problem using a heuristic approach.

Shobaki *et al*. [4] present a combinatorial algorithm for balancing RP and ILP during pre-allocation scheduling. In this combinatorial algorithm, the objective function is a weighted sum of RP and schedule length. The algorithm uses branch-and-bound enumeration with some pruning techniques that optimally schedule basic blocks with up to 664 instructions, which is substantially larger than the largest problem solved in previous work. With their combinatorial algorithm applied only to the hot functions, they report compiling the entire FP2006 benchmark suite in about 12 min (35% increase relative to the base compiler), which makes their algorithm the closest combinatorial algorithm to being practically acceptable. However, about 17% of the basic blocks are not solved to optimality. These blocks that time out are naturally the larger blocks, which, unfortunately, have higher impacts on the execution time. Furthermore, an inherent limitation of the algorithm of Shobaki *et al*. is that the RP component of the cost function is the peak excess register pressure, which is the maximum excess register pressure at any point in the schedule. The excess register pressure at a given point in the schedule is the difference between the RP at that point and the number of physical registers on the target processor. As explained in the paper of Shobaki *et al.*, this cost function does not always correlate well with the amount of spill code. A schedule that has the lowest peak pressure may have high pressure at so many nonpeak points that it results in more spills than a schedule with a larger peak but fewer high-pressure points.

In conclusion, there have been many attempts to solve this problem using both heuristic and combinatorial approaches, but none of these attempts has been shown to give an optimal or near-optimal solution to the problem.

4. HEURISTIC TECHNIQUES

The heuristic techniques studied in this paper are based on *list scheduling*. List scheduling is perhaps the most commonly used approach to instruction scheduling in production compilers [3]. It is based on maintaining a *ready list* of instructions. An instruction is ready if the instructions that it depends on have been scheduled. The instructions in the ready list are ranked according to certain *priority schemes*, and at each step in the algorithm, the instruction with the highest priority is scheduled. This makes list scheduling a greedy algorithm. In this paper, we consider the following priority schemes for ranking instructions in the ready list:

1. The instruction's *critical path* (CP) distance, which is the length of the longest weighted path in the DDG between the instruction and the end of the DDG [3]. This is a commonly used

DOI: 10.1002/spe

priority scheme that has been shown experimentally to give near-optimal results in scheduling a basic block for the sole objective of minimizing the number of cycles (pure ILP scheduling) [11].

2. The instruction's *last use count* (LUC). The LUC of an unscheduled instruction x at a given point during the construction of a schedule is the number of virtual registers that x is the last user of. In other words, the LUC of an instruction is the number of open live ranges that will be closed if instruction x is scheduled. The intuition behind this priority scheme is that favoring the instruction that closes the maximum number of open live ranges will tend to shorten live ranges and hence minimize the overlap among them. It is noted here that most instructions have either one or two input operands. Accordingly, the number of open live ranges that an instruction may close will be one or two in most cases.

When multiple priority schemes are used to rank the instructions in the ready list, the order in which the priorities are applied (the permutation of priority schemes) is an important factor that determines the performance of the resulting schedule. The idea of favoring an instruction if it is the last user of an open live range has been mentioned in the literature (for example, Cooper and Torczon [3]). The LUC priority scheme described in the current paper is a quantification of this idea. Furthermore, to the authors' best knowledge, the performance impact of this priority scheme and its interaction with other priority schemes using different permutations has not been studied experimentally. In this paper, we present a quantitative experimental study of all possible combinations and permutations of the CP and LUC priority schemes and show that one particular permutation of these two schemes produces better results than LLVM's best heuristic. The combinations and permutations studied in this paper are as follows:

CP	critical path is the only priority scheme.
LUC	last use count is the only priority scheme.
CP_LUC	critical path is the primary scheme and last use count is the secondary scheme.
LUC_CP	last use count is the primary scheme and critical path is the secondary scheme.

When there is a tie, that is, two or more instructions have the same value for the primary and secondary priority schemes, a tie breaking scheme (or a final priority scheme) is needed. In the experimental evaluation of this paper, ties are broken according to the instruction number assigned by the LLVM Compiler. Because it is not clear whether this instruction number is assigned according to any specific attribute, it will be assumed in the analysis that ties are broken arbitrarily.

It is noted here that the impact of the secondary priority scheme depends on the probability of ties on the primary priority scheme. In a large basic block, LUC ties are much more likely than CP ties, because a typical instruction has one or two operands, which limits the value of the LUC to zero, one, or two, while the CP distance has many possible values (depending on the block size and complexity). Therefore, we expect to see a substantial difference in performance between LUC and LUC_CP, while the difference in performance between CP and CP_LUC is expected to be relatively small. The experimental results will confirm this expectation.

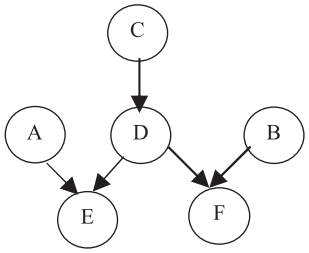
4.1. Example

Consider the DDG shown in Figure 1(a). For simplicity, it is assumed that the target machine can issue one instruction per cycle (single-issue machine). Instructions A, B, C, and D define virtual registers r_A, r_B, r_C, r_D, respectively. Instructions E and F are assumed to be store instructions that do not define any registers. It is assumed that all dependencies in the DDG are data (true) dependencies with unit latencies. Ties are broken according to the alphabetical order of instruction names.

Figure 1(b) shows the schedule produced by the CP heuristic. Instruction C has a CP distance of 2, Instructions A, D, and B have CP distances of 1, and Instructions E and F have CP distances of 0. Accordingly, Instruction C is scheduled in Cycle 1. Next, Instructions A, D, and B have the same CP distance of 1. Because there is no secondary priority scheme, the tie will be broken according to the alphabetical order (A, B then D) as shown in Figure 1(b). Finally, Instructions E and F, which have the same CP distance of 0, will be scheduled in alphabetical order. Figure 1(b) shows the set of live registers (Live Regs) after scheduling each instruction and the RP at each point. The peak RP in this case is 3, and this peak value occurs at two different points in the schedule (Cycles 3 and 4).

DOI: 10.1002/spe

(a) DDG



(b) CP heuristic schedule

Cycle	Instr.	Live Regs	RP
1	C	r_C	1
2	A	r_C, r_A	2
3	B	r_C, r_A, r_B	3
4	D	r_A, r_B, r_D	3
5	E	r_B, r_D	2
6	F	None	0

(c) CP_LUC heuristic schedule

Cycle	Instr.	Live Regs	RP
1	C	r_C	1
2	D	r_D	1
3	A	r_D, r_A	2
4	B	r_D, r_A, r_B	3
5	E	r_D, r_B	2
6	F	None	0

(d) LUC heuristic schedule

Cycle	Instr.	Live Regs	RP
1	A	r_A	1
2	B	r_A, r_B	2
3	C	r_A, r_B, r_C	3
4	D	r_A, r_B, r_D	3
5	E	r_B, r_D	2
6	F	None	0

(e) LUC_CP heuristic schedule

Cycle	Instr.	Live Regs	RP
1	C	r_C	1
2	D	r_D	1
3	A	r_D, r_A	2
4	E	r_D	1
5	B	r_D, r_B	2
6	F	None	0

Figure 1. Scheduling example (a) DDG; (b) CP heuristic schedule; (c) CP_LUC heuristic schedule; (d) LUC heuristic schedule; (e) LUC_CP heuristic schedule.

DOI: 10.1002/spe

Figure 1(c) shows the schedule obtained using the CP_LUC heuristic; that is, when the LUC is used as a secondary heuristic after CP. As in the CP-only case, Instruction C has the largest CP and will be scheduled first. In this case, however, the CP tie among Instructions A, D, and B will be broken according to the LUC. Instruction D has an LUC of 1, because it is the last user of live register r_C, while Instructions A and B have LUC values of zero, because they do not use any live registers. Accordingly, Instruction D is scheduled first, thus closing the r_C live range and opening the r_D live range. Then, Instructions A and B are scheduled in Cycles 3 and 4, raising the RP to 3. Finally, Instructions E and F are scheduled.

It is noted here that although the CP and CP_LUC schedules have the same peak RP, the peak value occurs at two points in the CP case, while it only occurs at one point in the CP_LUC case. In a large basic block, having high RP at fewer points is likely to result in fewer spills. As shown in the experimental results, the CP_LUC produces fewer spills, on average, than the CP heuristic.

Figure 1(d) shows the schedule produced by the LUC heuristic. Initially, all ready instructions (A, B, and C) have an LUC of 0. Therefore, the three instructions are scheduled in alphabetical order, thus raising the RP to 3. After scheduling Instruction C, Instruction D is scheduled because it is the only ready instruction. Instruction D closes the r_C live range and opens the r_D live range, thus keeping the RP at 3. Then, Instructions E and F are scheduled in alphabetical order, because they both have the same LUC of 2. This schedule also has a peak RP of 3.

Finally, Figure 1(e) shows the schedule produced by the LUC_CP heuristic. In this case, the LUC tie among Instructions A, B, and C is broken according to the CP value, which gives priority to Instruction C that has a CP distance of 2. Then, Instruction D is scheduled, because it has an LUC of 1 (it closes live range r_C), while the other two ready instructions (A and B) have an LUC value of 0. Then, Instructions A and B have the same LUC of 0 and the same CP distance of 1. This tie will be broken according to the alphabetical order, and Instruction A will be scheduled first, raising the RP to 2. After scheduling Instruction A, Instruction E will be ready with an LUC of 1 (it is the last user of live range r_A). Because LUC is the primary priority scheme, Instruction E will be scheduled before Instruction B, thus closing live range r_A and reducing the RP to 1. Finally, Instructions B and F will be scheduled. Unlike the other three schedules, this schedule has a peak RP of only 2. The experimental results of this paper will show that this is not a coincidence. Over a statistically significant set of instances, the LUC_CP heuristic produces substantially less spill code than the other three heuristics.

The previous example only considers RP reduction. In previous work, it has been shown experimentally that the CP priority scheme is a very effective scheme for scheduling a basic block with the sole objective of exploiting ILP. Shobaki [11] applies the CP heuristic to the basic blocks in CPU2000 targeting different machine models. He reports that more than 99% of the basic blocks are optimally scheduled for ILP using the CP heuristic. Even for the small percentage (less than 1%) of basic blocks that are not optimally scheduled, the difference between the CP schedule and the optimal schedule is not significant. These results suggest that adding CP as a secondary priority scheme after LUC is expected to achieve some degree of balance between RP and ILP. The key point illustrated by the previous example is that adding CP as a secondary priority scheme after the LUC priority scheme does not only achieve a better balance between RP and ILP but it also helps reduce RP. From a pure RP scheduling point of view, using CP as a secondary priority scheme produces schedules with lower RP, because it favors instructions that have longer dependence chains below them (Instruction C in the previous example), thus making it more likely to discover and favor instructions that close open live ranges. The experimental results of this paper will confirm that adding CP as a secondary priority scheme has a positive impact on both ILP and RP.

The previous example also shows that the CP priority scheme, when used as a primary priority scheme, tends to increase RP, because it favors scheduling instructions that start new dependence chains, thus opening new live ranges instead of closing open live ranges. Furthermore, because CP ties are unlikely in large complex basic blocks, adding the LUC as a secondary priority scheme after CP is not expected to make a big difference compared with using CP alone. The experimental results will confirm both of these expectations.

DOI: 10.1002/spe

5. CHALLENGES IN DEVELOPING AN OPTIMAL SOLUTION

The experimental results of this paper (Section 6) show that there is a large gap between the best heuristic considered in the study and the optimal solution. This raises a question about the possibility of developing an algorithm that gives the optimal solution in all cases. In this section, we discuss the challenges that are involved in developing such an algorithm, which may be developed using one of the following approaches:

The integrated approach. In this approach, scheduling and allocation are formulated as one optimization problem. This may also be viewed as a formulation of the register allocation problem that allows instruction *reordering*. In theory, this is the ideal approach. In practice, however, such a formulation will be quite challenging to develop. Even without reordering, there has been no formulation of optimal register allocation that was shown to allocate registers in real programs within reasonable time. An optimal register allocator has to make decisions about spilling, splitting, and coalescing that minimize the spill cost. Adding instruction reordering to the solution space will certainly make the problem much more complicated.

The decoupled approach. The complexity of the integrated approach has led researchers and practitioners to follow a decoupled approach, in which scheduling and allocation are formulated as two separate problems. Production compilers, such as GNU Compiler Collection (GCC), LLVM, and Open64, use the decoupled approach. The challenge in developing an optimal algorithm using this approach lies in finding a scheduling cost function that correlates well with the register allocator's objective, which is minimizing a certain spill cost. The allocator's spill cost is defined according to some cost model, which takes into account the number of needed loads and stores and their execution frequencies. Usually, the pre-allocation scheduler will not have access to all the information used in computing this cost. Even if this information is available, that will only strengthen the correlation between the scheduling objective and the allocation objective; it will not match the accuracy of the integrated approach in which there is only one cost function to minimize. This is an inherent limitation of the decoupled approach.

Considering ILP will make the problem even more complicated. Unlike the spill cost, which can be precisely estimated at compile time, ILP is hard to model at compile time, especially for complex out-of-order processors. Estimating ILP at compile time requires a precise machine model that simulates the processor's run-time behavior. Run-time behavior, however, depends on many factors, such as caching and conditional branching, that are hard to predict at compile time. For example, instruction scheduling for ILP is highly dependent on instruction latencies. Actual instruction latencies at run time depend on the state of the machine (the pipeline and the memory system, especially the cache), which in turn depends on previously executed instructions. Because instruction scheduling is performed on one code region (typically a basic block) at a time, the state of the machine at the region's entry point depends on the reaching code path, which is input dependent. In any nontrivial program with loops, there is generally an infinite number of code paths and consequently an infinite number of possible machine states (every iteration of a loop may lead to a different machine state). Different machine states lead to different latencies and hence different scheduling decisions. Determining the set of reaching paths and hence the set of possible machine states cannot, in general, be computed exactly at compile time; we can only predict the most likely paths using profile feedback.

The previous discussion shows that the complexity of the spill-code minimization problem is not limited to the fact that all combinatorial formulations of the problem are NP-hard. The problem of minimizing spill code in a modern production compiler is hard to formulate as an optimization problem that can be solved within reasonable time for real programs. It is hard to combine scheduling and allocation in one formulation, hard to find a scheduling objective that correlates well with the allocation objective, and hard to find a compile-time objective that correlates well with run-time performance.

6. EXPERIMENTAL EVALUATION

The scheduling heuristics described in this paper were implemented in the LLVM Compiler back end (Version 3.3) and evaluated using the SPEC CPU2006 benchmarks, targeting x86-64. Because

DOI: 10.1002/spe

CPU2006 has many FORTRAN programs, we used the GCC 4.6.3 as a front end with the Dragon Egg plug-in that supports FORTRAN. The LLVM back end has multiple pre-allocation scheduling heuristics. In this paper, we focus on the following two heuristics:

1. A bottom-up register-pressure-reduction (BURR) heuristic whose objective is minimizing register pressure based on the Sethi–Ullman numbers [12]. This heuristic is included in the study, because our experimental evaluation has shown that it gives the best overall performance among all LLVM heuristics from both spill code minimization and actual execution time points of view on x86-64.
2. An ILP heuristic whose objective is balancing ILP and register pressure. This ILP heuristic is used in this paper as a baseline, because in LLVM 3.3, it is the default heuristic on x86-64.

The tests were run on a machine having two Intel Xeon E5540 (Intel, Santa Clara, California, USA) processors running at 2.53 GHz with 24 GB of memory. Each CPU has eight threads (16 threads in total). All our tests, however, were compiled and executed using a single thread. In all tests, LLVM was invoked with the –O3 optimization option as well as the following performance tuning options: -march = core2, -mtune = core2. With these options, LLVM's default register allocator is used, which is a greedy global allocator in LLVM 3.3. The operating system is Ubuntu 10.10 (the 64-bit version).

6.1. Experimental methodology

A primary metric that is used in this paper to evaluate heuristic performance is a lower bound on the gap between the performance of the heuristic under study and optimal performance. This lower bound is computed by evaluating the difference between the sum of spills produced by each heuristic for a large set of functions and an *experimental upper bound* on the *optimal sum of spills* for this function set. The optimal sum of spills for a function set is the sum over the given function set of the spills generated by the register allocator if an optimal algorithm is used to schedule the basic blocks in this function set. An experimental upper bound on the optimal sum of spills for a large set of functions is computed by applying multiple heuristics to each function, taking the minimum number of spills per function across all heuristics (best result) and then summing these minima over all functions. For a given set of functions F and a given set of heuristics H, we define the sum of minima:

$$SOM(F,H) = \sum_{F} \underset{H}{Min}(spills(f,h)) \tag{1}$$

where spills(f, h) is the number of spills generated by the register allocator when heuristic $h \in H$ is used to schedule the basic blocks in function $f \in F$. The sum of minima (SOM) is an upper bound on the optimal sum of spills across the given function set F. That is because if an optimal algorithm for solving the pre-allocation scheduling problem exists, the number of spills produced by that optimal algorithm must be less than or equal to the number of spills produced by the best heuristic in any experimental study; otherwise, that algorithm will not be optimal from register pressure point of view.

To evaluate the performance of a given heuristic on a given set of functions F, we compute the difference between the sum of spills produced by that heuristic on F and the $SOM(F, H)$ taken over a set of heuristics H. For a given heuristic h in a set of heuristics H, we define the difference between h's spill sum and SOM as follows:

$$Dif(h,H) = \sum_{F} spills(f,h) - SOM(F,H) \tag{2}$$

$Dif(h, H)$ is a *lower bound* on the size of the gap between the number of spills produced by this heuristic and the optimal number of spills. Clearly, increasing the number of functions and the number of heuristics included in the experiment will increase the precision (tightness) of the upper and lower bounds that are computed by *SOM* and *Dif*, respectively.

It should be noted here that although the goal in this paper is studying the impact of instruction scheduling, the actual number of spills generated is determined by the register allocation algorithm. For any given schedule, different register allocation algorithms may make different spilling

DOI: 10.1002/spe

decisions. Given two schedules S1 and S2 and two register allocation algorithms A1 and A2, if A1 produces more spills with S1 than it does with S2, A2 will not necessarily produce more spills with S1 than it does with S2. So, generally speaking, the quality of a given schedule is relative to the register allocation algorithm. However, if a sufficiently large sample of basic blocks is scheduled using different scheduling algorithms and allocated using the *same* register allocation algorithm, large sampling is expected to neutralize the dependence on the register allocation algorithm. Thus, the total number of spills produced by each scheduling algorithm across this large sample will give a strong indication about the effectiveness of each scheduling algorithm at reducing register pressure.

6.2. Benchmark statistics

The benchmark suite used in our experiments is SPEC CPU2006, which has two groups of benchmarks: INT2006 containing 12 integer programs that are characterized by control-intensive code and FP2006 containing 17 floating-point programs[¶] that are characterized by compute-intensive code using many FP operations. Table I shows some interesting statistics about these benchmarks.

6.3. Spill code statistics

In this section, we evaluate the impact of various heuristics on the amount of spill code generated by the register allocator. The heuristics included in the evaluation are two of LLVM's heuristics (ILP and BURR), the four heuristics described in Section 4 (CP, CP_LUC, LUC, and LUC_CP), and an extension of the LUC_CP, in which each node's *successor count* (SC) is used as a third priority scheme if multiple candidates have the same LUC and the same CP. We refer to the latter heuristic as LUC_CP_SC. For all five heuristics, the node ID assigned by the LLVM compiler to each instruction was used as the final tie breaker. Furthermore, the evaluation covers the combinatorial scheduling algorithm described in our previous work [4]. The combinatorial algorithm is studied using two different heuristics for computing the initial feasible schedule: BURR and LUC_CP. We refer to these algorithms as COMB_LUC_CP and COMB_BURR, respectively. For a fair comparison with LLVM's heuristics, the combinatorial algorithm is studied using LLVM's rough latencies. With these latencies, a large register pressure weight (RPW) does not make sense. Therefore, an RPW of 1 was used. The time limit was set to 10 ms per instruction.

In addition to the previous heuristics, four other heuristics were included in the experimental evaluation, but their results are not shown in the paper. These heuristics are LLVM's *source* heuristic, LLVM's *fast* heuristic, LUC_SC_CP (SC used as a second priority scheme), and SC_LUC_CP (SC used as a primary priority scheme). In overall performance, LLVM's *source* and *fast* heuristics were outperformed by LLVM's BURR heuristic, while LUC_SC_CP and SC_LUC_CP were outperformed by LUC_CP_SC. The relatively poor performance of these heuristics did not justify using larger tables to report their results. However, these omitted heuristics were used in computing the SOM in Table II so that the tightest possible upper bound on optimal behavior is used in the evaluation.

Table I. CPU2006 statistics.

	INT2006	FP2006
Number of benchmarks	12	17
Number of functions	30,260	17,188
Number of basic blocks	405,224	399,736
Avg. basic blocks per function	13.4	23.3
Avg. instructions per basic block	9.7	15.7
Max. instructions per basic block	614	3946

[¶]The dealII benchmark did not compile with GCC 4.6.3 because of a syntactic issue. So, it was excluded in all experiments. Libquantum was also excluded because of the large random variation in its execution time.

DOI: 10.1002/spe

Each heuristic was applied to all functions in CPU2006, and the number of live ranges spilled by LLVM's default register allocator was used as a metric for evaluating the scheduling heuristic's effectiveness at reducing register pressure.

Table II shows spilling statistics for CPU2006. Row 1 shows the total number of functions in each of INT2006 and FP2006. Row 2 shows the number of functions that have spills. A function may have spills with some scheduling heuristics but not with other heuristics. The entry in Row 2 is the number of functions that have spills with *at least* one scheduling heuristic. The entry in Row 3 (SOM) was computed using Equation (1) for all the heuristics included in the study. Row 4 shows the average spills per function (Row 3 divided by Row 1), and Row 5 shows the average spills per spilling function (Row 3 divided by Row 2). The numbers in Table II show that the FP2006 benchmarks generally have much more spills than the INT2006 benchmarks. This is consistent with the fact that FP benchmarks generally have more computation.

Tables III and IV show the number of spilled ranges produced by each heuristic relative to the SOM listed in Table II. The second column (extra spills) in each table shows the difference between the total spills produced by each heuristic and the SOM. For example, the ILP heuristic produces 1696 spills more than the SOM on INT2006, which corresponds to a percentage difference of 8.7%. As mentioned in Section 6.1, this difference is a lower bound on the gap between the heuristic solution and the optimal solution to this problem; that is, the ILP heuristic produces at least 8.7% more spills than an optimal algorithm. With this metric, the best heuristic on INT2006 is COMB_BURR (the combinatorial algorithm with BURR used to generate the initial feasible schedule), which produces 5.2% more spills than the minimum. The second best heuristic is COMB_LUC_CP (the combinatorial algorithm with LUC_CP used to generate the initial feasible schedule), which produces 8.1% more spills than the minimum. Interestingly, the LUC_CP heuristic used alone produces approximately the same number of extra spills as COMB_LUC_CP. It is also noted that LUC_CP_SC and BURR produce approximately the same number of extra spills as LUC_CP on INT2006. The worst performance on INT2006 is that of LUC, CP, and CP_LUC. This shows that the worst results are obtained if no second priority scheme is used, thus frequently making random tie breaking (which is the case in LUC) or when CP is used as a primary priority scheme (which is the case in CP and CP_LUC).

On FP2006 (Tables IV), the best heuristic according to the extra-spills metric is BURR, which produces 4.5% more spills than the minimum, followed by COMB_BURR, which produces

Table II. Spilling statistics for INT2006 and FP2006.

		INT2006	FP2006
1	Total functions	30,260	17,188
2	Functions with spills	3262 (10.8%)	7221 (42.0%)
3	Sum of minima (SOM)	19,452	255,414
4	Avg. spills per function	0.64	14.9
5	Avg. spills per spilling function	5.96	35.4

Table III. Total spilled ranges produced by each heuristic in INT2006.

Heuristic	Extra spills	%Funcs at min	Max extra per func
ILP	1696 (8.7%)	77.0%	113 (150) 75%
BURR	1606 (8.3%)	77.7%	113 (150) 75%
CP	1869 (9.6%)	73.1%	103 (150) 69%
CP_LUC	1712 (8.8%)	75.2%	104 (150) 69%
LUC	2389 (12.3%)	68.6%	91 (346) 26%
LUC_CP	1584 (8.1%)	75.4%	74 (346) 21%
LUC_CP_SC	1602 (8.2%)	75.6%	74 (346) 21%
COMB_LUC_CP	1576 (8.1%)	75.4%	74 (346) 21%
COMB_BURR	1021 (5.2%)	84.9%	80 (150) 53%

DOI: 10.1002/spe

Table IV. Total spilled ranges produced by each heuristic in FP2006.

Heuristic	Extra spills	%Funcs at min	Max extra per func
ILP	21,660 (8.5%)	55.7%	1086 (2322) 47%
BURR	11,460 (4.5%)	60.9%	172 (2322) 7%
CP	40,543 (15.9%)	47.0%	2079 (2322) 90%
CP_LUC	36,622 (14.3%)	49.8%	2149 (2322) 93%
LUC	37,235 (14.5%)	38.6%	924 (0) ∞
LUC_CP	19,069 (7.4%)	55.0%	1118 (2322) 48%
LUC_CP_SC	18,762 (7.3%)	54.8%	1144 (2322) 49%
COMB_LUC_CP	17,817 (7.0%)	55.3%	1205 (2322) 52%
COMB_BURR	13,746 (5.4%)	66.3%	1274 (2322) 55%

5.4% more spills than the minimum. The third best heuristic is COMB_LUC_CP with 7% extra spills. The LUC_CP_SC and the LUC_CP heuristics rank fourth and fifth with 7.3% and 7.4% extra spills, respectively. These two heuristics then have comparable performance to that of COMB_LUC_CP. Similar to the INT2006 results, the FP2006 results show that CP, CP_LUC, and LUC have the poorest register pressure reduction performance. On FP2006, there is a fairly large gap in the amount of spilling between these three poorly performing heuristics and the rest of the heuristics. It is also noted that CP_LUC consistently has a better register pressure reduction performance than CP.

The third column (%Funcs at min) shows the percentage of functions for which each heuristic produced the minimum number of spilled ranges; that is, the percentage of functions for which this heuristic was the best heuristic (possibly tying with other heuristics). According to this metric, the best heuristic on both IN2006 and FP2006 is COMB_BURR, which achieves the minimum spill count for 85% of the INT2006 functions and 66% of the FP2006 functions. The second best heuristic is BURR, which achieves the minimum spill count on 78% of the IN2006 functions and 61% of the FP2006 functions. The third best heuristic is ILP with 77% on INT2006 and 56% on FP2006. The three related heuristics LUC_CP, LUC_CP_SC, and COMB_LUC_CP have comparable results around 75% on INT2006 and around 55% on FP2006.

The last column (Max extra per func) shows the maximum number of extra spills that each heuristic produced in a single function relative to the minimum number of spills for that function. The minimum number of spills for that function is shown between parentheses, followed by the percentage difference in spills between the heuristic under study and the best heuristic for that function. This metric is intended to give an indication about each heuristic's worst-case performance. The numbers in Tables III and IV show that all heuristics may produce significant amounts of extra spills on some functions. On FP2006, all heuristics, except BURR and LUC, produce over a thousand extra spills on a single function. It is noted here that BURR and LUC are the only two heuristics that do not take ILP into account. This suggests that taking ILP into account degrades a heuristic's worst-case behavior.

It is interesting to note that all heuristics, except LUC, exhibit their worst-case behavior on one particular function in FP2006 (all but one of the numbers in parenthesis in the last column are equal). This function is from the Wrf benchmark. Interestingly, the heuristic that gives the minimum amount of spilling (2322 spilled ranges) on this function is the LUC heuristic, which has the worst overall performance according to most metrics in the table. In fact, the LUC entry in the last column shows that LUC produced 924 spills in a function for which some other heuristic produced zero spills. This shows that all heuristics may produce a significant amount of unnecessary spills on some functions. If these unnecessary spills occur in a hot function, that will lead to a significant performance degradation.

Most metrics in Tables III and IV suggest that the most effective register pressure reduction technique in the studied set is COMB_BURR, followed by BURR. This is an expected result, because BURR is LLVM's register-pressure-specific heuristic, and using that heuristic to generate the initial feasible schedule for a combinatorial scheduler that searches for the best possible schedule within a given time limit should generally give the best results. The extra-spills metric on FP2006, however,

DOI: 10.1002/spe

suggests that BURR produces slightly less spill code than COMB_BURR. This is attributed to the limitation of the peak excess register pressure cost function used by the combinatorial scheduler (detailed in Section 3), as well as the fact that the combinatorial scheduler does not always complete its search within the given time limit. Furthermore, we note that BURR's only objective is minimizing register pressure, while in COMB_BURR, the objective is balancing register pressure and ILP. The execution-speed results in the next section will show that minimizing spilling without accounting for ILP does not necessarily lead to the best overall performance. So, although BURR produces slightly less spilling than the combinatorial scheduler, the combinatorial scheduler gives better execution time on average.

The results in Tables III and IV show that the LUC_CP permutation gives significantly better results than all other combinations and permutations of the LUC and CP priority schemes. Comparing the results of LUC_CP with those of LUC alone shows that using CP as a secondary priority scheme has a significant impact on register pressure reduction. Using the LUC priority scheme by itself produces 14.5% more spills than the SOM on FP2006, while using the LUC_CP permutation produces only 7.4% more spills than the SOM. As explained in Section 4, this is attributed to the fact that when no instruction has an advantage over other instructions in closing open live ranges, a good choice will be the instruction with the maximum CP. That choice will start the longest dependence chain as early as possible, thus giving more options at subsequent steps for closing live ranges in this dependence chain (see the example in Section 4).

It is also noted that, in spite of its simplicity, LUC_CP has good register-pressure-reduction performance compared to LLVM's heuristics; it even produces significantly less spilling than the ILP heuristic on FP2006. The LUC_CP_SC heuristic, which adds a third priority scheme to the two schemes used in LUC_CP has approximately the same performance as LUC_CP; it produces slightly more spills on INT2006 and slightly fewer spills on FP2006. COMB_LUC_CP, which is the combinatorial algorithm with LUC_CP used to generate the initial feasible schedule, produces fewer spills than LUC_CP on both INT2006 and FP2006. The results in the next section will show that the LUC_CP heuristic and its derivatives (LUC_CP_SC and COMB_LUC_CP) give the best overall run-time performance.

Comparing heuristics' success rates at achieving minimal spilling (the "%Funcs at min" column in Tables III and IV) on FP2006 with those on INT2006 shows that the success rates on FP2006 are much lower (the maximum success rate is 66% on FP2006 and 85% on INT2006). These low success rates reflect the relative difficulty of the scheduling problem on the FP programs, which, on average, have larger basic blocks with higher register pressures.

Overall, these results show that there is a large gap between the best heuristic and optimal behavior. On FP2006, BURR produces 4.5% more spills than the SOM. It produces the minimum amount of spilling on only 61% of the functions, which implies that for *at least* 39% of the functions, BURR produces unnecessary spills. As mentioned in Section 6.1, including more heuristics in the experiment will result in even tighter lower bounds on the size of the gap between the best heuristic and the optimal solution. So, we expect that BURR actually produces extra spills in more than 39% of the functions.

6.4. Execution speed

In this section, we evaluate the impact of each heuristic on the execution speed. Tables V and VI show for each heuristic the percentage increase in the SPEC score relative to LLVM's default heuristic on x86-64, which is the ILP heuristic. A SPEC score represents speed rather than time (a larger score indicates faster execution). Therefore, positive numbers in Tables V and VI indicate performance improvements, and negative numbers indicate performance degradations. The results for LUC_CP_SC are not shown, because they are, to a very good approximation, the same as the results for LUC_CP.

In running SPEC tests, random variations may occur. This is due to the random factors that affect a program's performance. To minimize the impact of this random variation on our experimental evaluation, each test was run nine times. The SPEC running utility automatically takes the median

DOI: 10.1002/spe

score for each benchmark. Each entry in Tables V and VI is the percentage difference between the median score for the heuristic under study and the median score for the base heuristic (LLVM's default). When the calculated percentage difference is less than 1%, it is considered uncertain (random noise) and the table entry is set to zero. Of course, some of these ignored differences may be real differences, but the possibility of random variation leads to a high degree of uncertainty about such small differences; hence, the elimination. For one particular benchmark, namely Libquantum, random variation was so high that we decided to exclude that benchmark from our experimental study.

It is noted here that although the percentage differences are listed in the table using a single-decimal digit, the geometric means were computed using higher precision. We note also that although a percentage difference less than 1% on an individual benchmark is considered uncertain, a percentage difference of less than 1% in the computed geometric mean may be fairly certain and meaningful. Consider for example a heuristic that improves the performance of one benchmark in FP2006 relative to the base heuristic by 10%, which is quite certain. If this heuristic does not change the performance of any other benchmark in FP2006, this will give a geometric-mean improvement of 0.56% on FP2006. Clearly, this 0.56% is a certain and meaningful difference in the geometric-mean, because it is calculated based on a highly certain score difference.

Table V. Percentage increase in execution speed relative to LLVM's default heuristic in INT2006.

Benchmark	LLVM BURR	CP	CP_LUC	LUC	LUC_CP	COMB LUC_CP	COMB BURR
Perlbench	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Bzip2	0.0%	1.9%	2.5%	0.0%	1.9%	1.9%	0.0%
GCC	0.0%	0.0%	−3.4%	0.0%	0.0%	0.0%	0.0%
MCF	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Gobmk	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Hmmer	2.2%	0.0%	2.9%	2.2%	2.9%	2.9%	−2.2%
Sjeng	0.0%	−1.0%	−1.5%	−2.0%	−2.0%	−1.0%	0.0%
H264ref	0.0%	−11.5%	−8.1%	−1.0%	0.0%	0.0%	0.0%
Omnetpp	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.9%
Astar	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Xalancbmk	0.0%	0.0%	1.3%	1.3%	0.0%	0.0%	0.0%
Geo-mean	**0.32%**	**−1.01%**	**−0.50%**	**0.15%**	**0.37%**	**0.46%**	**−0.03%**

Table VI. Percentage increase in execution speed relative to LLVM's default heuristic in FP2006.

Benchmark	LLVM BURR	CP	CP_LUC	LUC	LUC_CP	COMB LUC_CP	COMB BURR
Bwaves	0.0%	0.0%	0.0%	0.0%	2.2%	1.6%	0.0%
Gamess	0.0%	−2.5%	0.0%	−4.6%	0.0%	0.0%	0.0%
Milc	0.0%	−1.7%	−1.1%	0.0%	1.7%	1.7%	1.1%
Zeusmp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.4%
Gromacs	12.5%	5.4%	3.6%	2.7%	8.0%	8.9%	14.3%
CactusADM	0.0%	−7.3%	−5.1%	0.0%	5.1%	3.7%	2.2%
Leslie	0.0%	1.0%	1.6%	0.0%	0.0%	1.6%	0.0%
Namd	0.0%	−3.1%	−1.9%	RE[a]	0.0%	0.0%	0.0%
Soplex	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Povray	−1.88%	−1.9%	−1.4%	0.0%	0.0%	0.0%	0.0%
Calculix	6.0%	0.0%	0.0%	14.3%	10.9%	10.1%	7.0%
GemsFDTD	1.6%	4.8%	3.7%	1.6%	2.7%	3.2%	1.6%
Tonto	0.0%	1.2%	1.2%	0.0%	1.8%	1.8%	0.0%
Lbm	30.0%	30.7%	31.4%	10.7%	31.4%	32.8%	33.8%
Wrf	0.0%	0.0%	0.0%	0.0%	1.2%	1.2%	0.0%
Sphinx3	0.0%	−3.5%	−1.7%	−6.6%	1.4%	1.4%	1.0%
Geo-mean	**2.76%**	**1.16%**	**1.63%**	**1.09%**	**3.89%**	**3.98%**	**3.59%**

[a]A run-time error occurred.

DOI: 10.1002/spe

To better study the relation between the amount of spill code and execution time, Tables VII and VIII show the sum of *spill costs* in each benchmark's hot functions. The spill cost in a given function is a weighted sum of the store and load instructions inserted by the register allocator with each load or store multiplied by an estimate of its execution frequency. The execution frequency takes the loop nesting level into account. While the spilled-range count reported in Tables III and IV is believed to be a good metric for evaluating a heuristic's register pressure reduction capability, the spill cost is a better metric for studying the impact of each heuristic on execution time. Hot function selection was the same as that made in the authors' previous work [4] based on the profiling data published by Weicker and Henning [13]. According to these criteria, 57 functions in INT2006 and 65 functions in FP2006 were selected. The last row in Tables VII and VIII shows the sum of hot spill costs across INT2006 and FP2006, respectively. The number in parentheses in the last row is the percentage increase in the hot spill cost sum relative to the base heuristic (LLVM's ILP).

Comparing the spilled-range information in Tables III and IV with the hot spill costs in Tables VII and VIII shows a correlation for some but not all heuristics. For example, the information in both tables suggests that CP and CP_LUC have poor register-pressure-reduction performance. On the other hand, according to Table IV, BURR produces the minimum total amount of spilling on FP2006, while the information in Table VIII suggests that LUC_CP produces the minimum total amount of spilling. The difference is attributed to the fact that the set of hot functions is a small subset of the total function set (there are tens of thousands of functions in CPU2006, out of which, only tens of functions are hot). Although the set of hot functions is the set that controls the benchmarks' execution speeds, this set is not statistically significant enough to accurately assess a heuristic's overall register pressure reduction performance. Therefore, we use the hot spill costs in Tables VII and VIII to analyze the heuristics' impacts on execution speeds and use the spilled-range information in Tables III and IV to assess the heuristics' register-pressure-reduction capabilities.

Tables VII and VIII show that most benchmarks have spills in their hot functions. Mcf and Milc are the only two benchmarks that do not have any spills in their hot functions. The FP benchmarks generally have more spills than the INT benchmarks, which is an expected result, because FP benchmarks have larger basic blocks involving more intensive computation.

Examining the execution speeds in Table V shows that although the impact of instruction scheduling on the performance of most INT2006 benchmarks is limited, scheduling has a significant impact on the performance of some benchmarks. In particular, the performance of H264ref is highly sensitive to scheduling. The results for this benchmark show that the heuristics that have CP as a primary priority scheme (CP and CP_LUC) produce poor schedules that lead to significant performance degradations relative to the base heuristic. Examining the amount of hot spills in H264ref (Table VII) shows an obvious strong correlation between performance degradations and increases in hot spill costs. The hot spill cost with the CP heuristic is 37% higher than the base heuristic cost, and that causes an 11.5% degradation in speed. With the CP_LUC heuristic, the hot spill cost is 21% higher than the base, and that causes an 8.1% degradation in speed. This result shows that the impact of scheduling on performance is not limited to floating-point benchmarks; integer benchmarks may also be significantly impacted by scheduling if they have high register pressure in their hot code. The rest of the discussion in this section focuses on the FP2006 benchmarks.

The results in Table VI show that the best overall performance on FP2006 is achieved using COMB_LUC_CP and LUC_CP (the difference between the two heuristics is negligibly small) although, according to the results in Section 6.3, these two heuristics do not have the best register pressure reduction performance. COMB_BURR has the third best performance followed by LLVM's BURR. CP, CP_LUC and LUC have poor execution-speed performance that correlates with their register-pressure-reduction performance. The fact that the ranking of the first four heuristics in execution speed does not correlate with their ranking in Tables III and IV or Tables VII and VIII suggests that register pressure reduction is not the only factor that controls performance. The best overall execution speed on FP2006 is achieved by COMB_LUC_CP and LUC_CP, which balance register pressure and ILP. It is interesting to note that although COMB_BURR has a significantly better register pressure reduction performance than COMB_LUC_CP as suggested by Tables III, IV, VII and VIII, COMB_LUC_CP gives a better overall execution time. These results

DOI: 10.1002/spe

Table VII. Hot spill costs in INT 2006 with 57 hot functions.

Benchmark	LLVM ILP	LLVM BURR	CP	CP_LUC	LUC	LUC_CP	COMB LUC_CP	COMB BURR
Perlbench	5325	5325	4947	5218	4926	4889	4889	4961
Bzip2	16,887	16,869	15,860	15,776	15,713	15,713	15,713	5930
GCC	1585	1585	1585	1585	1585	1585	1585	1235
MCF	0	0	0	0	0	0	0	0
Gobmk	120,074	122,027	115,418	116,429	124,144	118,868	118,868	127,900
Hmmer	941	874	941	864	874	864	864	1060
Sjeng	676	671	706	744	2244	696	696	586
H264ref	105,955	106,633	144,748	128,330	100,728	104,675	104,675	103,951
Omnetpp	3	3	3	3	3	3	3	3
Astar	1665	1665	1665	1665	1842	2189	2189	1665
Xalancbmk	110	110	110	110	110	110	110	101
Sum	**253,221**	**255,762 (1%)**	**285,983 (12.9%)**	**270,724 (6.9%)**	**252,169 (−0.4%)**	**249,592 (−1.4%)**	**249,592 (−1.4%)**	**247,392 (−2.3%)**

DOI: 10.1002/spe

Table VIII. Hot spill costs in FP 2006 with 65 hot functions.

Benchmark	LLVM ILP	LLVM BURR	CP	CP_LUC	LUC	LUC_CP	COMB LUC_CP	COMB BURR
Bwaves	6,115,224	6,492,717	6,678,743	7,243,809	5,449,915	5,029,210	6,270,324	6,458,801
Gamess	716,090	414,416	570,462	544,546	774,879	529,757	529,757	453,994
Milc	0	0	0	0	0	0	0	0
Zeusmp	115,170	98,815	140,896	117,292	123,187	114,945	112,847	97,748
Gromacs	13,347	11,276	15,709	15,361	12,642	12,905	12,314	13,558
CactusADM	969,158	667,005	1,379,911	1,176,520	664,644	734,140	769,262	810,334
Leslie	248,762	209,703	212,908	219,211	225,032	216,865	223,256	200,922
Namd	79,557	93,059	103,786	99,105	82,138	78,769	74,894	77,370
Soplex	322	324	314	314	316	316	316	413
Povray	889	889	1546	1546	889	889	889	889
Calculix	1,700,131	1,704,153	3,165,214	3,164,181	3,043,573	2,286,363	2,286,364	1,705,359
GemsFDTD	632,960	585,618	574,541	585,762	634,783	587,699	580,521	579,821
Tonto	274,357	271,549	280,328	278,454	271,640	277,631	277,641	274,003
Lbm	1119	90	70	70	282	60	60	60
Wrf	135,259	131,730	142,182	136,311	139,810	132,349	135,645	131,442
Sphinx3	161	161	151	151	161	132	132	161
Sum	**11,002,506**	**10,681,505 (−2.9%)**	**13,266,761 (20.6%)**	**13,582,633 (23.5%)**	**11,423,891 (3.8%)**	**10,002,030 (−9.1%)**	**11,274,222 (2.5%)**	**10,804,875 (−1.8%)**

DOI: 10.1002/spe

show that, contrary to the assumption made in previous work [1, 2], compiler scheduling for ILP can in some cases have a significant impact on performance even on out-of-order machines.

Examining the performance numbers of LUC_CP in Tables V and VI shows that this heuristic is a well-balanced heuristic that does not cause any significant degradation relative to the base heuristic (there is only a 2% degradation on Sjeng). Compared with all other heuristics, the LUC_CP heuristic is significantly outperformed only on two benchmarks: Gromacs (outperformed by BURR and COMB_BURR) and Calculix (outperformed by LUC). Given this balanced performance and the simplicity of the heuristic, LUC_CP appears to be a good choice in practice. The results for LUC_CP_SC (not shown in Tables V, VI, VII and VIII) are almost identical to those of LUC_CP, which indicates that adding the SC as a third priority scheme does not make a significant difference.

On the other end of the spectrum, CP, which is a pure ILP heuristic that tends to increase register pressure, has poor run-time performance with a geometric-mean improvement of 1.16% relative to the base heuristic. On individual benchmarks, CP causes substantial performance degradations on H264ref (11.5%) and Cactus (7.3%). Looking at the hot spill costs for these two benchmarks in Tables VII and VIII shows that these degradations are consistent with substantial increases in the hot spill cost. Because CP is a heuristic that tends to increase register pressure, the relatively poor run-time performance of CP on x86 confirms the logical expectation that register pressure reduction should be the primary objective of pre-allocation scheduling on out-of-order machines.

Comparing CP with CP_LUC shows that using LUC as a secondary priority scheme leads to a significant but not substantial improvement in overall performance. On FP2006, the geometric mean is improved from 1.16% to 1.63%. On individual benchmarks, CP_LUC gives significantly better performance on H264ref, Gamess, and Cactus. Looking at the hot spill costs for these three benchmarks in Tables VII and VIII shows that these performance improvements correlate very well with spill cost reductions in these benchmarks' hot functions. On H264ref, for example, using LUC as a secondary priority scheme reduces the hot spill cost from 144,748 to 128,330, which leads to reducing the performance degradation relative to the base heuristic from 11.5% to 8.1%.

Explaining the performance of the LUC heuristic is particularly difficult compared with other heuristics, because, as explained in Section 4, LUC ties are much more likely than CP ties, and LUC breaks these ties arbitrarily. This arbitrary tie breaking leads to a high degree of randomness in this heuristic's behavior. For example, this heuristic happens to produce the lowest hot spill costs on two benchmarks (H264ref and Cactus) and the highest hot spill costs on four benchmarks (Sjeng, Gamess, Gems, and Tonto).

One particularly interesting case in the previous tables is the Lbm benchmark. This benchmark has one dominant hot function in which the program spends 99% of its time. That function has one large basic block that dominates its performance as well as a few small basic blocks. Comparing the execution speed in Table VI with the hot spill costs in Table VIII shows a strong correlation between the amount of hot spills and the execution speed. All heuristics produce significantly less spilling than the base heuristic, and that leads to significantly faster execution (up to 33.8%). LUC reduces hot spills by 75%, which leads to a performance improvement of 10.7%, while all other heuristics reduce hot spills by 92% to 95%, and that leads to performance improvements around 30%. These results show how substantial the impact of pre-allocation scheduling on a program's performance can be.

6.5. Spill code density

Table IX shows spill code density information for all functions and hot functions. The spill code density in a function is the number of spill instructions divided by the number of instructions in the function. The spilling information shown in the table is for LLVM's BURR heuristic. With this heuristic, the register allocator generates, on average, 2.73 spill instructions per spilled range in INT2006 and 1.87 spill instructions per spilled range in FP2006. FP2006 benchmarks have much more spills than INT2006. In both benchmark suites, hot functions have significantly more spills than other functions. On FP2006, the spill code density is 0.079 spills per instruction across the entire benchmark suite, while the spill code density across hot functions is 0.177 spills per instruction.

DOI: 10.1002/spe

Table IX. Spill code density for hot functions and all functions.

	INT2006	FP2006
Instructions per function	130.0	365.8
Spills per function	1.91	28.99
Spill code density per function	0.0147	0.079
Instructions per hot function	1029.6	1901.3
Spills per hot function	34.7	336.6
Spill code density per hot function	0.034	0.177

This is an expected result, because a hot function tends to have more computation and consequently higher register pressure. Recall that the two factors that determine whether a given function is hot are the function's frequency of execution and the function's execution time (the amount of computation performed by the function). A function's frequency of execution has nothing to do with register pressure inside the function but the amount of computation performed by the function correlates with register pressure; more computation tends to lead to higher register pressure. The numbers in Table IX are experimental confirmations of this fact.

6.6. Compile times

The total compile time of CPU2006 with the LUC_CP heuristic is approximately equal to the compile time with LLVM's BURR and ILP heuristics. However, our current prototype implementation of the LUC_CP heuristic involves some overhead, and we believe that further refinement of the implementation is possible.

7. CONCLUSIONS AND FUTURE WORK

This paper presents an experimental study using SPEC CPU2006 on x86-64 of various pre-allocation scheduling heuristics. Based on this experimental study, the paper proposes a simple but effective heuristic (LUC_CP) for doing pre-allocation instruction scheduling on an out-of-order processor. The proposed heuristic gives the best overall performance among all the heuristics in the study. The proposed heuristic is a straightforward heuristic that works well without any case-specific tuning. The results of the study, however, show that even the best heuristic can produce excessive spills in many cases, which leads to the conclusion that it is unlikely to find a heuristic that works well in all cases.

Developing a pre-allocation scheduling technique that works well in all cases will require a rigorous approach that formulates the problem as a combinatorial optimization problem. For the reasons explained in the paper, this is a very challenging task. We are currently exploring multiple combinatorial formulations of this problem.

ACKNOWLEDGEMENTS

The authors thank the Computer Science (CS) Department at PSUT for providing the machine that was used to perform this experimental evaluation. We specially thank the system and network administration team at PSUT including Mohammad Abusaad, Husam Abed, and Waseem F. Faous for the technical support they provided to us. We also thank the anonymous reviewers for their constructive comments that led to significantly improving the final version of this paper.

REFERENCES

1. Govindarajan R, Yang H, Amaral J, Zhang C, Gao G. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Transaction Computers* 2003; **52**(1): 4–20.
2. Barany G, Krall A. Optimal and heuristic global code motion for minimal spilling. In *Proc. Intl. Conf. on Compiler Construction*, Italy, 2013.
3. Cooper K, Torczon L. Engineering a Compiler. *Morgan Kaufmann*, 2004.

DOI: 10.1002/spe

4. Shobaki G, Shawabkeh M, Abu-Rmaileh N. Pre-allocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 2013; **10**(3): 14:1–14:31. Article 14.
5. Goodman J, Hsu W. Code scheduling and register allocation in large basic blocks. In *Proc. Int'l Conf. Supercomputing*, 1988.
6. Faraboschi P, Fisher J, Young C. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE* 2001; **89**(11): 1638–1659.
7. Berson D, Gupta R, Soffa M. URSA: a unified resource allocator for registers and functional units in VLIW architectures. In *Proc. IFIP Working Conf. on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 1993; 243–254.
8. Touati S. Register saturation in instruction-level parallelism. *International Journal of Parallel Programming* 2005; **33**(4): 393–449.
9. Kessler C. Scheduling Expression DAGs for Minimal Register Need. Computer Languages, 1998.
10. Malik A. Constraint programming techniques for optimal instruction scheduling. Ph.D thesis, University of Waterloo, 2008.
11. Shobaki G. Optimal global instruction scheduling using enumeration. Ph.D dissertation, Department of Computer Science, UC Davis, 2006.
12. Sethi R, Ullman JD. The generation of optimal code for arithmetic expressions. *Journal of the ACM* 1970; **17**(4): 715–728.
13. Weicker R, Henning J. Subroutine Profiling Results for the CPU2006 Benchmarks. *ACM SIGARCH Computer Arch. News* 2007.

DOI: 10.1002/spe