Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach

GHASSAN SHOBAKI, Princess Sumaya University for Technology
MAXIM SHAWABKEH, Google
NAJM ELDEEN ABU RMAILEH, Princess Sumaya University for Technology

Balancing Instruction-Level Parallelism (ILP) and register pressure during preallocation instruction scheduling is a fundamentally important problem in code generation and optimization. The problem is known to be NP-complete. Many heuristic techniques have been proposed to solve this problem. However, due to the inherently conflicting requirements of maximizing ILP and minimizing register pressure, heuristic techniques may produce poor schedules in many cases. If such cases occur in hot code, significant performance degradation may result. A few combinatorial optimization approaches have also been proposed, but none of them has been shown to solve large real-world instances within reasonable time. This article presents the first combinatorial algorithm that is efficient enough to optimally solve large instances of this problem (basic blocks with hundreds of instructions) within a few seconds per instance. The proposed algorithm uses branch-and-bound enumeration with a number of powerful pruning techniques to efficiently search the solution space. The search is based on a cost function that incorporates schedule length and register pressure. An implementation of the proposed scheduling algorithm has been integrated into the LLVM Compiler and evaluated using SPEC CPU 2006. On x86-64, with a time limit of 10ms per instruction, it optimally schedules 79% of the hot basic blocks in FP2006. Another 19% of the blocks are not optimally scheduled but are improved in cost relative to LLVM's heuristic. This improves the execution time of some benchmarks by up to 21%, with a geometric-mean improvement of 2.4% across the entire benchmark suite. With the use of precise latency information, the geometric-mean improvement is increased to 2.8%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems—*Sequencing and scheduling*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*

General Terms: Algorithms, Performance, Design, Experimentation, Languages

Additional Key Words and Phrases: Compiler optimizations, optimal instruction scheduling, instruction-level parallelism (ILP), register pressure reduction, branch-and-bound enumeration, NP-complete problems, performance optimization

ACM Reference Format:
Shobaki, G., Shawabkeh, M., and Abu Rmaileh, N. E. 2013. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. ACM Trans. Architec. Code Optim. 10, 3, Article 14 (September 2013), 31 pages.
DOI: http://dx.doi.org/10.1145/2512432

Authors' addresses: G. Shobaki (corresponding author), Princess Sumaya University for Technology, Khalil Saket Street, Al-Jubaiha Amman 11941, P.O. Box 1438, Jordan; email: g.shobaki@psut.edu.jo; M. Shawabkeh, Google; N. E. Abu Rmaileh, Princess Sumaya University for Technology, Khalil Saket Street, Al-Jubaiha Amman 11941, P.O. Box 1438, Jordan.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1544-3566/2013/09-ART14 $15.00
DOI: http://dx.doi.org/10.1145/2512432

1. INTRODUCTION

Instruction scheduling and register allocation are two important compiler optimizations. The objective of instruction scheduling is exploiting Instruction-Level Parallelism (ILP) by overlapping the execution of independent instructions, while the objective of register allocation is allocating as many program variables as possible to CPU registers. Variables that cannot be allocated to registers are *spilled* to main memory. Both instruction scheduling and register allocation are known to be NP-complete [Cooper and Torczon 2004]. Furthermore, these two optimizations have conflicting requirements, because overlapping the execution of more independent instructions requires more registers, thus increasing the need for spilling.

Many algorithms have been proposed to balance the two conflicting objectives of exploiting ILP and minimizing spills. Some of these algorithms are based on an *integrated* approach, where scheduling and allocation are done simultaneously in one phase [Berson et al. 1993]. Other algorithms are based on a *decoupled* approach, in which scheduling and allocation are done in two separate phases, but the scheduling phase that precedes register allocation (*preallocation scheduling*) tries to minimize register requirements in order to reduce the amount of spill code generated in the allocation phase. Due to the complexity of integrating scheduling and allocation in one phase, most production compilers adopt the decoupled approach [Cooper and Torczon 2004]. Most proposed algorithms for doing preallocation scheduling are based on heuristics [Goodman and Hsu 1988; Govindarajan et al. 2003], which may produce suboptimal solutions in many cases. If such cases happen to occur in hot code, significant performance degradation may result. Combinatorial algorithms have also been proposed for preallocation scheduling [Kessler 1998; Malik 2008; Barany and Krall 2013]. However, none of these algorithms offers a complete solution for balancing ILP and register pressure in real-world instances with large sizes.

This article presents a combinatorial algorithm for doing preallocation scheduling with the objective of balancing ILP and register pressure. The proposed algorithm is efficient enough to optimally schedule large basic blocks with hundreds of instructions and improve the heuristic schedules of blocks with thousands of instructions.

It is important to emphasize that the use of the word "optimal" to describe the schedules generated by the proposed combinatorial algorithm should not be interpreted as a claim of generating code with optimal performance. In this article preallocation scheduling is formulated as a combinatorial optimization problem, and an optimal schedule is an optimal solution to this combinatorial optimization problem. As thoroughly explained in the article, that optimal schedule does not always lead to optimal runtime performance. A program's runtime performance depends on a combination of many factors. Due to the complexity of the interactions among these factors, the compilation process is divided into multiple *phases* (passes), and each optimization phase addresses a small subset of these factors. In a typical production compiler, preallocation instruction scheduling is an optimization phase that addresses two factors affecting performance: ILP and register pressure. The contribution of the current article is showing that formulating preallocation instruction scheduling as an optimization problem and using a combinatorial optimization approach to solve it can, in many cases, produce much better performance than using a heuristic approach. In solving a compiler optimization problem with multiple conflicting objectives, a greedy heuristic is likely to produce inferior solutions for the larger and more complex instances, because a heuristic does not backtrack to undo bad decisions. A combinatorial approach, on the other hand, has the ability to backtrack, and this enables it to discover solutions that are very hard to discover by a heuristic. The experimental results of this article show that the proposed combinatorial optimization approach

does indeed generate code with significantly better runtime performance for some real programs.

The combinatorial technique used in this article is branch-and-bound enumeration, which exhaustively explores the solution space. As detailed in Section 4, the solution space is explored by traversing a *decision tree* whose internal nodes represent partial schedules and whose leaves represent complete schedules. The challenge in applying branch-and-bound enumeration to a compiler optimization problem lies in developing efficient pruning techniques that make it possible to explore the solution space with minimal increase in compile time. This work uses pruning techniques from previous work on combinatorial scheduling [Shobaki et al. 2009; Shobaki and Wilken 2004; Shobaki 2006], as well as new pruning techniques that are developed specifically for register pressure minimization.

To control the increase in compile time, a heuristic schedule is first constructed. Then a lower bound technique is applied to check the heuristic schedule for optimality. If the heuristic schedule is not optimal, the branch-and-bound enumerator is invoked and given a certain time limit to find a better schedule. For most instances, the enumerator will find better schedules, and, depending on the complexity of the instance, it may or may not converge into a provably optimal schedule. As shown in the experimental results, there are many instances for which the enumerator does not find an optimal schedule but still finds a schedule that is significantly better than the heuristic schedule. From a practical point of view, this approach is more effective than the approaches that require optimality.

The combinatorial algorithm presented in this article can also be used as a performance analysis and tuning tool, where longer time limits may be given to maximize the chances of finding an optimal schedule. The resulting optimal schedule can then be used by compiler developers to assess and enhance their scheduling heuristics.

The contributions of this article can be summarized as follows.

—We extend previous work on combinatorial instruction scheduling using enumeration [Shobaki 2006; Shobaki and Wilken 2004; Shobaki et al. 2009] to devise an efficient combinatorial scheduling algorithm that also minimizes register pressure. To the authors' best knowledge, the algorithm presented in this article is the first combinatorial algorithm that balances ILP and register pressure and can optimally solve instances with hundreds of instructions within a few seconds per instance.
—We provide a retargetable preallocation scheduling algorithm. What distinguishes the proposed algorithm from previous algorithms is that it is not inherently biased towards ILP or register pressure. Either ILP or register pressure can be made the primary objective by adjusting the cost function. Thus, the proposed algorithm can be used for multiple targets, ranging from VLIW machines, where ILP is likely to be the primary objective, to modern out-of-order superscalar machines, where register pressure is likely to be the primary objective.
—We describe a framework for achieving a compromise between the speed of heuristic scheduling and the precision of combinatorial scheduling by starting with a heuristic schedule and giving the enumerator a certain time limit to incrementally improve the heuristic schedule.
—Unlike most previous work on combinatorial scheduling, this work studies the practicality of applying combinatorial scheduling to a production compiler. The proposed algorithm has been integrated into the LLVM Compiler and its performance and compile time are evaluated using the SPEC CPU2006 benchmarks on real x86 hardware. For several benchmarks, the algorithm gives significant performance improvements compared to LLVM's heuristic scheduler with minimal increase in compile time. To

the authors' best knowledge, the work presented in this article is the first successful attempt to integrate a combinatorial scheduler into a production compiler and achieve significant performance improvements on real hardware using the current standard benchmark suite with a reasonable increase in compile time.

—We evaluate the implicit assumption made in previous work that minimizing register requirements during preallocation scheduling leads to minimizing the amount of spill code generated by the register allocator. This article emphasizes the subtle difference between these two objectives and studies the degree of correlation between them.

The rest of the article is organized as follows. Section 2 defines the problem. Section 3 summarizes relevant previous work. Section 4 gives a detailed description of the proposed algorithm, and Section 5 explains the algorithm's limitations. Section 6 presents the experimental results, and Section 7 summarizes the article's contribution and outlines future work.

2. BACKGROUND AND PROBLEM DEFINITION

The problem addressed in this article is preallocation instruction scheduling with the objective of finding a schedule that achieves the best possible balance between minimizing schedule length and minimizing register pressure. This section precisely defines this objective.

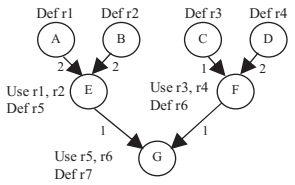
Given a sequence of instructions in a program's basic block with their dependencies represented by a Data Dependence Graph (DDG), an instruction schedule is an assignment of instructions to machine cycles. The *schedule length* is the estimated number of cycles needed to execute the instruction stream. The number of cycles is estimated based on a certain *machine model*. A machine model is a specification of the functional units available on the target machine, a mapping between instructions and functional units, as well as the number of functional unit instances that are available in each cycle. The machine model also specifies the latency of each instruction and whether this instruction is pipelined or not. The proposed algorithm and its implementation support a general machine model with arbitrary functional unit and latency information. The experimental evaluation, however, uses LLVM's simple machine model for x86, which assumes a single-issue machine with one generic functional unit.

In the preallocation scheduling phase, registers in the code are virtual registers, but, in some special cases, the code may contain physical registers. Each register has a specific data type. Given an instruction schedule, the *Register Pressure* (RP) for a given data type at a given point in the schedule is the number of registers of that type that are live at that point.

Various cost functions can be used to represent register pressure. The primary cost function used in this article is the *Peak Excess Register Pressure* (PERP) in the given basic block. The *Excess Register Pressure* (ERP) of a given data type at a given point in a schedule is the difference between the register pressure at that point and the number of physical registers that are available of that data type (physical register limit). The PERP is the maximum value of the ERP across the schedule. When registers of multiple data types are used, the *total* ERP at a given point is the sum of the ERPs of all data types at that point, and the schedule's total PERP is the maximum total ERP across the schedule.

Register pressure computation is based on the Def and Use sets of the scheduled instructions. The Def set of an instruction is the set of registers that are defined (produced) by that instruction, and the Use set is the set of registers that are used (consumed) by that instruction. A typical binary arithmetic instruction, such as ADD, uses two registers and defines one register. The general formulation of this article allows an instruction to have an arbitrary number of Defs and Uses.

(a) Data Dependence Graph (DDG)



(b) CP list schedule and its ERP, assuming 2 physical registers.

Cycle	Instr.	Live Regs	RP	ERP
1	A	r1	1	0
2	B	r1, r2	2	0
3	D	r1, r2, r4	3	1
4	C	r1, r2, r4, r3	4	2
5	E	r4, r3, r5	3	1
6	F	r5, r6	2	0
7	G	r7	1	0

Fig. 1. Data dependence graph and register pressure.

Register pressure computation is illustrated by the example in Figure 1. Figure 1(a) shows a DDG with 7 instructions. A node represents an instruction and an edge represents a dependency, with the edge weight representing the latency. The Def and Use sets for each instruction are shown next to the corresponding DDG node. In this simple example, it is assumed that the target machine can issue one instruction per cycle (single-issue machine) and that all registers are of the same data type. Figure 1(b) shows a feasible schedule for the given DDG. The schedule is constructed using Critical-Path (CP) list scheduling [Cooper and Torczon 2004]. The schedule length is 7 cycles, which is clearly optimal from ILP point of view, because there are no empty cycles.

The third column in Figure 1(b) shows, for each scheduled instruction, the registers that are live after that instruction is issued. For example, after issuing Instruction A, Register r_1 is live. After issuing Instruction E in Cycle 5, Registers r_1 and r_2 are no longer live, because Instruction E is the last use of each of these two registers. The corresponding RP is shown in Column 4 and the ERP is shown in Column 5. The ERP is calculated assuming that the target machine has 2 physical registers of the given type. The PERP is the maximum ERP, which is 2 (reached at Cycle 4).

Given a schedule S, with schedule length $|S|$, the cost of this schedule is defined as

$$Cost(S) = |S| - L + wP, \tag{1}$$

where L is a lower bound on the schedule length, P is the *register pressure cost function*, and w is the *Register Pressure Weight* (RPW). The primary register pressure cost function used in this article is the PERP. However, alternate register pressure cost functions have been considered in the research, as discussed in Section 5. The parameter w expresses the weight of register pressure relative to the schedule length. When $w = 1$, a PERP of 1 is as costly as one extra cycle. The objective of the proposed combinatorial scheduling algorithm is to find a schedule that minimizes this cost function. The weight w is used to bias the algorithm towards minimizing either schedule length or register pressure. This can be used to adjust the cost function for the target machine. For example, on a target machine with out-of-order execution and a small number of registers, a larger value for w may be used to make register pressure minimization the primary objective.

3. PRIOR WORK

There is a large volume of research on instruction scheduling and register allocation that relates to our work. In this section, we cover the most relevant work on register pressure reduction in preallocation scheduling, focusing on the combinatorial algorithms that are comparable to our work.

Heuristic solutions to this problem have been proposed by Goodman and Hsu [1988], Berson et al. [1993], Touati [2005], Govindarajan et al. [2003], and most recently by Barany [2011, 2013]. Combinatorial solutions have been presented by Govindarajan et al. [2003], Kessler [1998], Malik [2008], and Barany and Krall [2013].

Berson et al. [1993] propose an integrated approach to scheduling and allocation that unifies both problems into one resource allocation problem. In this approach, both registers and functional units are treated as resources. The technique constructs a Directed Acyclic Graph (DAG) that measures resource requirements and identifies the regions that have excessive resource requirements. Excessive requirements are either high degrees of parallelism or high register pressures. Certain heuristics are then used to reduce these excessive requirements by adding edges to the DAG to introduce sequentiality, or equivalently, reduce parallelism.

Barany [2011] presents a technique, called *Register Reuse Scheduling,* that reduces the 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 register pressure. Barany's technique has the advantage of using the *global* spill costs that are computed by the register allocator, as opposed to using preallocation estimates of register requirements. Although this leads to a stronger correlation between the scheduling objective and the spill cost, the selection of live ranges for register reuse is done using a greedy heuristic. In a more recent paper, Barany and Krall [2013] propose a combinatorial algorithm using integer linear programming for doing live range selection. This algorithm has the additional advantage of allowing global code motion across basic blocks. However, both algorithms (combinatorial and heuristic) ignore ILP and schedule for the sole objective of reducing register pressure. Ignoring ILP is based on the assumption that modern out-of-order machines can exploit ILP without the compiler's help. As shown in our experimental results, this assumption is true for most but not all benchmarks. ILP is an important factor in some computation-intensive programs.

Barany and Krall's [2013] experimental results show large increases in compile times and only small improvements in execution times. They report a total compile time of 18 hours for 20 SPEC CPU2000 benchmarks and a geometric-mean improvement of 0.5% in execution time. The maximum improvement in a single benchmark is only 3%. These modest improvements in execution times may be due to timeouts on larger functions as well as Barany and Krall's exclusion of functions with more than 1000 instructions. As shown in our experimental results, many hot functions are fairly large and the larger basic blocks within these functions tend to have higher impacts on actual execution times.

Govindarajan et al. [2003] present both a heuristic technique and a combinatorial technique. Their main contribution is a heuristic technique for minimizing register pressure alone, ignoring ILP. Similar to Barany and Krall's [2013] technique, ignoring ILP may lead to degrading performance on some computation-intensive programs. To evaluate their heuristic technique, Govindarajan et al. implemented a combinatorial algorithm using integer linear programming. Their combinatorial algorithm, however, could only solve DDGs with an arithmetic mean of 19 instructions per DDG.

Kessler [1998] presents a combinatorial register pressure reduction algorithm based on a combination of branch-and-bound enumeration and dynamic programming.

Instead of building a decision tree, the algorithm builds a decision DAG (called the *Selection DAG* in the paper). The nodes of this DAG represent partial schedules (solutions to scheduling subproblems), and the algorithm identifies common subproblems and equivalent subproblems (permutations of the same set of instructions) to minimize the number of DAG nodes. The idea of detecting equivalent subproblems is similar to the history-based pruning technique described in Section 4.2 of this article but with a different implementation that requires more space.

In spite of the pruning techniques that are used to minimize DAG nodes, Kessler's algorithm [1998] still generates a large number of nodes, and that leads to exhausting memory on larger instances. Kessler's register pressure algorithm runs out of space and time on DDGs with more than 50 instructions. Kessler also extends the algorithm to account for ILP but at a high computational cost in both space and time. The spatial cost is so high that the ILP-aware version of the algorithm runs out of space for DDGs with more than 25 instructions. Kessler's ILP-aware algorithm does not use any lower bound techniques like Rim and Jain's [1994] or Langevin and Cerny [1996].

Malik [2008] presents a combinatorial algorithm using constraint programming to solve the problem of basic block scheduling without spilling, that is, finding a minimum-length schedule with a PERP of zero, if such a schedule exists. This problem has limited practical value, because our experimentation shows that many FP benchmarks have spill code in their hot functions even on x86-64. Malik's algorithm solves DDGs with up to 50 instructions within 10 minutes per DDG.

The algorithm proposed in the current article is built on our previous work on combinatorial instruction scheduling using enumeration [Shobaki 2006; Shobaki and Wilken 2004; Shobaki et al. 2009]. The objective in our previous work was limited to exploiting ILP without accounting for register pressure. In the current work, the objective is extended to include register pressure. Adding register pressure minimization significantly increases the difficulty of the problem. The results reported in previous work suggest that scheduling for minimum register pressure is harder than scheduling for ILP and that balancing the two objectives is even harder. From a combinatorial optimization point of view, adding register pressure to the cost function greatly increases the size of the solution space. It is important to note here that the combinatorial explosion of the register pressure reduction problem can only be handled by developing efficient algorithms that aggressively prune the solution space; running on faster hardware will not mitigate the exponential explosion.

Unlike Kessler's algorithm [1998], the algorithm proposed in this article is based on a strict depth-first traversal of the decision tree, in which only one path is active at any given time. This ensures that the number of active tree nodes will never exceed the schedule length, thus limiting the amount of space needed. If the history-based pruning technique described in Section 4.2 is disabled, the proposed algorithm uses a fixed amount of memory that does not increase with time. Therefore, the algorithm may be run for an arbitrarily long period of time without exhausting memory. Even when history-based domination is enabled, the increase in memory usage is slow enough to allow the algorithm to run for long periods of time on a typical machine. In our experimentation, we ran tests that lasted for multiple weeks and only used a small portion of the system's memory.

In addition to providing a register pressure reduction algorithm, the current article studies the practicality of applying a combinatorial optimization technique to a production compiler. The first attempt to integrate a combinatorial scheduler into a production compiler was made by Winkel [2007]. Winkel developed an integer linear programming algorithm for doing global instruction scheduling without explicitly accounting for register pressure. He integrated his scheduler into the Intel Compiler for Itanium and achieved significant performance improvements on a selected subset of the INT2006

benchmarks. Since Winkel's scheduler did not account for memory performance, the selected subset in his experiment included only five benchmarks. In terms of compile time, Winkel's scheduler was orders of magnitude slower than the heuristic scheduler and he used a time limit of 4 hours per function.

4. ALGORITHM'S DESCRIPTION

The proposed algorithm is an extension of the basic block scheduling algorithm described in Shobaki's dissertation [2006]. The basic block scheduling algorithm is extended by adding register pressure minimization. The extensions include generalizing the framework and developing register-pressure-specific pruning techniques. This section describes the new algorithm focusing on the extensions that have been added to achieve register pressure minimization.

The algorithm first constructs an initial heuristic schedule and then computes a lower bound on the schedule length using Langevin and Cerny's algorithm [Langevin and Cerny 1996]. The cost of the heuristic schedule is then computed using Equation (1). If the cost is zero, the heuristic schedule is optimal from both ILP and register pressure points of view and the algorithm terminates. If the cost is nonzero, the enumerator is invoked to search for a schedule with a lower cost. At that point, the heuristic schedule is the *current best schedule* and its cost is the *current best cost*.

The enumerator explores one schedule length at a time starting at the schedule-length lower bound. The schedule length being explored is referred to as the *target schedule length*. For each target schedule length, the enumerator searches for a feasible schedule with cost less than the current best cost. If a zero-cost schedule is found at the current target length T, that schedule is optimal and the algorithm terminates. Otherwise, the next target length $T + 1$ is considered. A schedule with length $T + 1$ can have a lower cost than the current best cost C_{min} only if the value of $T + 1 - L$ is less than C_{min} (see Equation (1)). If that condition is not satisfied, no better schedule can exist at the next target length $T + 1$, and the algorithm terminates.

Note that for a given target schedule length, the schedule-length component of the cost is fixed, and only the register pressure component varies among different feasible schedules. The search at each target schedule length is done by incrementally constructing a schedule by adding one instruction or a stall at a time. The search can be represented by a decision tree, or an *enumeration tree*, whose root represents an empty schedule. After scheduling an instruction, the enumerator branches from the current tree node to a new tree node. Each node in the tree then represents a *partial schedule*, and each leaf node represents a *complete schedule*.

To efficiently explore the tree, the enumerator applies at each node a number of *feasibility tests* and *cost tests* that may result in pruning the subtree below that node and backtracking to the previous node. A feasibility test checks if there is a feasible schedule with the target length below the current tree node, while a cost test checks if there is a feasible schedule with a lower cost than the current best cost. If any feasibility or cost test fails, the enumerator undoes the last decision (unschedules the last instruction scheduled) and backtracks to the previous tree node, where it attempts another candidate instruction, and so on.

The feasibility tests applied at each node are range tightening and relaxed scheduling, which are the same tests described in previous work on basic block scheduling for ILP [Shobaki 2006]. Relaxed scheduling uses Rim and Jain's algorithm [Rim and Jain 1994] to compute a lower bound on the schedule length for unscheduled instructions and compares that lower bound to the number of remaining cycles to determine if the target schedule length is still feasible. The cost tests, on the other hand, are the register pressure cost test and the history-based domination test, which are described next.

4.1. Register Pressure Cost Test

The enumerator keeps track of the register pressure and evaluates the register pressure cost at each tree node. If the register pressure cost function is the PERP, the cost at the current node is a valid lower bound on the cost below the current node, because the PERP is a monotonically increasing function. The cost lower bound is compared to the current best cost. If the cost lower bound at a given tree node is not less than the current best cost, no better schedule can be found below that node and the enumerator backtracks to consider other options.

To evaluate the cost, the enumerator incrementally computes the register pressure at each tree node. After scheduling an instruction, the enumerator examines the Def and Use sets of that instruction. Each virtual register defined by the instruction is added to the live set, and each virtual register used by the instruction decrements the number of unscheduled uses of that register. If the number of unscheduled uses of a given register reaches zero, that register is no longer live and is therefore removed from the live set. The RP at each tree node is the size of the live set at that node, and the ERP for each register type is the difference between the RP and the physical register limit for that register type.

4.2. History-Based Domination Test

History-based domination is a pruning technique that was introduced by Shobaki and Wilken to do efficient combinatorial instruction scheduling using branch-and-bound enumeration [Shobaki and Wilken 2004]. That work, however, was limited to scheduling for ILP without accounting for register pressure. Depending on the register pressure cost function, that technique may or may not apply to scheduling with register pressure minimization. In this section, we briefly summarize history-based domination, and then generalize it to derive a valid pruning technique for scheduling with register pressure minimization when the register pressure cost function is the PERP. For a detailed description of the original technique and its underlying data structures, the interested reader is referred to Shobaki's dissertation [2006].

The idea in this pruning technique is to store information about previously enumerated tree nodes in a *history table* and use this information to prove, under certain conditions, that the current tree node cannot have a lower-cost feasible schedule below it. The current node is compared with previously enumerated *similar* nodes. Two nodes are similar if their partial schedules are permutations of the same subset of instructions. When scheduling for minimum register pressure, the comparison between the two similar nodes must involve the following two tests.

Feasibility Test. In this test, the constraints on the remaining subproblem below the current node are compared to the constraints below the history node. If the history node is not more constrained than the current node, then every partial schedule that is feasible below the current node was *necessarily* a feasible partial schedule below the history node. In this case, it is said that the history node *dominates* the current node. The constraints that are compared to prove domination are latency and resource constraints. If the history node dominates the current node and no feasible schedule was found below the history node, then no feasible schedule can be found below the current node and the enumerator backtracks. If, on the other hand, the history node dominates the current node but there was a feasible schedule below the history node, the cost test is applied to check if the current node may have a lower-cost schedule.

Cost Test. This test is dependent on the cost function. For some cost functions, such a test may not exist. If the register pressure cost function is the PERP, it can be shown that if the history node dominates the current node in the feasibility test, it will be

sufficient to check the costs of the two partial schedules at the current node and the history node. If the cost of the history node's partial schedule is less than or equal to the cost of the current node's partial schedule, no lower-cost schedule can exist below the current node. The following argument shows why this is the case when the cost function is the PERP.

Consider a current tree node x with partial schedule P_x and a similar history node y with partial schedule P_y, such that y dominates x and the cost of P_y is less than or equal to the cost of P_x. A complete schedule C_x below x consists of a prefix partial schedule P_x above x and a suffix partial schedule S_x below x. The cost of C_x is the maximum of the *prefix cost* (PERP of P_x) and the *suffix cost* (PERP of S_x treated as a separate schedule). Similarly, a complete schedule C_y below y consists of a prefix partial schedule P_y above y and a suffix partial schedule S_y below y, and the cost of C_y is the maximum of the prefix cost (PERP of P_y) and the suffix cost (PERP of S_y). The PERP of a suffix partial schedule S_x below x depends on the permutation of the instructions (instruction order) within S_x and the *liveness status* at *x, that is,* which registers are live and which registers are not. The key idea is that the liveness status at x does not depend on the instruction order in P_x; it only depends on the *subset* of instructions scheduled in P_x. By definition, the similarity of nodes x and y implies that both P_x and P_y have the same subset of instructions scheduled and therefore the same liveness status. Furthermore, the fact that y dominates x implies that every instruction order that is feasible below x was also feasible below y. It follows that the best suffix cost below x cannot be any better than the best suffix cost below y. Since the prefix cost above y is less than or equal to the prefix cost above x, x cannot have any complete schedule with a lower cost than the best complete-schedule cost below y. Therefore, there is no complete schedule below x with cost less than the current best cost, and node x can be pruned away.

Note that the preceding argument does not apply to *any* cost function. For example, an interesting cost function that may correlate well with the amount of spill code is the sum of live range lengths in the schedule. For this cost function, the cost of the suffix partial schedule depends on the instruction order in the prefix partial schedule, and therefore the preceding argument does not apply to this cost function.

4.3. Complete Example

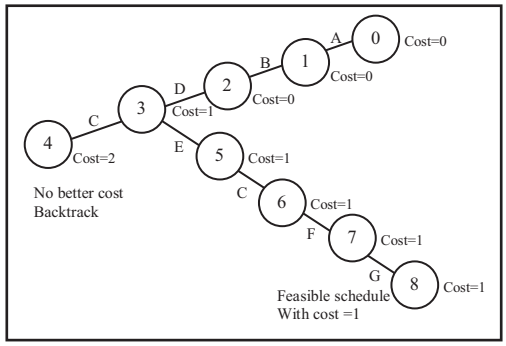
In this section the proposed combinatorial algorithm is illustrated by applying it to the DDG of Figure 1. It is assumed that all registers in the DDG are of the same type and that the machine has 2 physical registers of that type. The target machine is a single-issue machine and the register pressure weight (w in Equation (1)) is equal to 1.

For a general DDG, Langevin and Cerny's algorithm [1996] is used to compute a lower bound on the DDG's schedule length. In this simple example, however, 7 cycles is an obvious lower bound on a single-issue machine. Assuming that the initial heuristic schedule is the CP list schedule shown in Figure 1(b), the cost of that schedule is computed by substituting a schedule length of 7, a lower bound of 7, and a PERP of 2 into Equation (1).

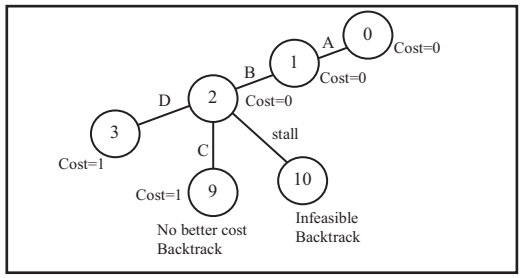
$$Cost(S_{\text{heur}}) = 7 - 7 + 1 \times 2 = 2$$

The best cost is then set to 2, and the enumerator is invoked to search for a schedule with cost less than 2. The enumerator first searches for a schedule with length 7 (the lower bound). The enumeration tree is shown in Figure 2. Note that the cost at each node is the PERP of the node's partial schedule, not the register pressure at the node itself. Assuming that the enumerator uses the critical-path heuristic for selecting instructions, it will first take the leftmost path in the tree of Figure 2(a) and construct the partial schedule <A, B, D, C>. After scheduling Instruction C, the register pressure

(a) finding the first feasible schedule



(b) feasibility and cost tests at Node 2



(c) history-based domination at Node 0

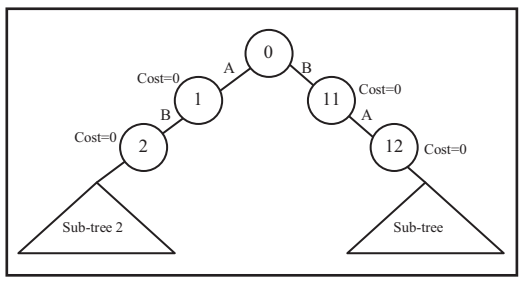


Fig. 2. Enumeration tree for the DDG in Figure 1.

is 4 and the cost, which is the PERP, is 2. Since 2 is equal to the best cost, the cost test will cause the enumerator to backtrack to Node 3.

At Node 3, another option is scheduling Instruction E instead of C. The enumerator then schedules Instruction E and takes the path that leads to the complete feasible schedule <A, B, D, E, C, F, G> with cost = 1 at Leaf Node 8. That schedule is then stored as the best schedule and the best cost is updated to 1. At that point, the enumerator does not know if that schedule is optimal. So, it backtracks to Node 2, since it is the deepest tree node with cost less than 1.

As shown in Figure 2(b), there are two other options at Node 2: scheduling Instruction C or scheduling a stall. If Instruction C is scheduled, the cost becomes 1, which is equal to the current best cost, and the enumerator backtracks. Scheduling a stall, on the other hand, cannot lead to a feasible schedule at target length 7. In this

simple example, infeasibility is obvious, but, in the general case, infeasibility will be detected by the relaxed scheduling feasibility test. In that test, Rim and Jain's algorithm [1994] will compute a lower bound of 5 on the schedule length for unscheduled instructions. That lower bound will then be compared to the number of remaining cycles, which is 4 cycles for the current target length of 7 (at Node 10, 3 cycles have been used by Instructions, A, B and the stall). This comparison will detect infeasibility at Node 10.

The enumerator then backtracks to Node 1, and the same tests will be applied to quickly detect that there are no better schedules below that node. The enumerator finally backtracks to the root node (Node 0) and examines the options of scheduling Instructions B, C, or D in the first cycle instead of A. Figure 2(c) shows how history-based domination is used to rule out the option of scheduling Instruction B then Instruction A.

If B is scheduled in Cycle 1 and A in Cycle 2, checking the history table will detect similarity between the current node (Node 12) and Node 2 in the history table, because Partial Schedules <A, B> and <B, A> are permutations of the same subset of instructions. The domination condition will be checked, and Node 2 will be found to dominate Node 12. In this simple example, it is intuitively obvious that the remaining subproblem below Node 2 is not more constrained than the remaining subproblem below Node 12 (because both Instructions A and B have a latency of 2, and E is the only dependent of both instructions). For the more general case, a precise sufficient condition for domination is given in Shobaki's dissertation [2006]. Combining the fact that Node 2 dominates Node 12 with the fact that both nodes have the same cost of zero, Node 12 can be pruned away and the enumerator backtracks.

After completing the search at target length 7 with a best cost of 1, the enumerator examines the next target length of 8 for possible exploration. In this case, increasing the schedule length by 1 will increase the total cost by 1. So, even if a schedule with zero register pressure cost is found at target length 8, the total cost will still be 1, which is equal to the current best cost. Therefore, target length 8 does not need to be examined, and the algorithm terminates with the now provably optimal schedule <A, B, D, E, C, F, G>, whose cost is 1.

5. ALGORITHM'S LIMITATIONS

The objective of the proposed algorithm is finding the best possible balance between exploiting ILP and minimizing spill code using the decoupled approach to instruction scheduling and register allocation. As mentioned before, this approach is the most widely used approach in production compilers due to the complexity of integrating scheduling and allocation in one phase [Cooper and Torczon 2004]. Furthermore, the proposed algorithm assumes that preallocation scheduling is done *locally* within the basic block. Due to the complexities of *global* instruction scheduling that moves instructions across basic blocks [Faraboschi et al. 2001; Havanki et al. 1998; Shobaki et al. 2009; Winkel 2007], many production compilers, including LLVM, only implement local instruction scheduling. Both the decoupling of scheduling and allocation and the locality of scheduling have inherent limitations. The following subsections study the impact of these limitations on the proposed algorithm and briefly describe the attempts that we have made to minimize this impact.

5.1. First Limitation: Register Pressure Cost Function

The challenge in using the decoupled approach to scheduling and allocation lies in finding a scheduling cost function that correlates well with the amount of spill code. In this work we have considered multiple cost functions. The primary cost function considered was the PERP. The PERP reflects the *register requirement*, that is, the

minimum number of physical registers needed by a given schedule. However, in register allocation, the objective is not finding the minimum number of physical registers needed to accommodate all virtual registers. Rather, the number of physical registers is fixed and the objective is minimizing spills. If schedule x has a lower PERP than schedule y, the amount of spilling with schedule x is not necessarily less than that with schedule y. In spite of having a lower peak pressure, schedule x may have more conflicts than schedule y at nonpeak points in the schedule, thus causing more spills. Interestingly, in most previous work it is implicitly assumed that minimizing the PERP necessarily minimizes spill code.

In an attempt to achieve a stronger correlation between the scheduling objective and the amount of spill code, we considered and experimented with alternate cost functions, including:

—the sum of excess register pressures at all points in the block; and
—the sum of the PERP and the average register pressure across all points in the block.

Experimentally, none of these cost functions gave better overall results than the simple PERP cost function, although some of them gave better results on some benchmarks. Thinking of a fundamentally better cost function that achieves a much stronger correlation with the amount of spill code led to the conclusion that such a cost function has to be aware of the register allocator's actual needs instead of trying to predict these needs. That requires moving the scheduler into the register allocator, which is essentially taking an integrated approach rather than a decoupled approach to scheduling and allocation. The integrated approach is beyond the scope of the current article but is likely to be an interesting topic for future work.

Although the PERP cost function does not always give the best results on blocks that require many spills, it always gives optimal results on blocks for which there exist no-spill schedules and excellent results on blocks that have nearly-no-spill schedules. Due to the enormous size of the solution space for large basic blocks with high degrees of parallelism, such schedules are often hard to discover by heuristics. As shown in the experimental results, the proposed algorithm with the PERP cost function eliminates many hot spills and significantly improves the execution times of the benchmarks in which such blocks are found in hot code.

5.2. Second Limitation: Scheduling within a Basic Block

In many production compilers, including LLVM, register allocation is done *globally* for the whole function, while preallocation scheduling is done *locally* within a basic block. Thus, the global live-interval analysis and spill cost estimates that are used by the register allocator to make spilling decisions are not available to the preallocation scheduler. Since a local minimum does not always lead to a global minimum, minimizing the local PERP in a given basic block may not minimize the amount of spilling generated by the global register allocator for some functions. Minimizing register pressure within a block will always have a positive impact on the amount of spill code for functions having one dominant basic block. As shown in Section 6.8, such basic blocks do occur in real applications.

In conclusion, eliminating both limitations requires integrating the scheduler into the register allocator so that the scheduler can take advantage of the global information on which the allocator bases its decisions. Barany reports significant improvements in spill cost by reordering instructions during register allocation [Barany 2011].

In spite of the aforementioned limitations, the experimental results of the next section show that the proposed algorithm in its current state with the PERP cost function can still produce significant performance improvements for many programs.

6. EXPERIMENTAL RESULTS

The combinatorial scheduling algorithm described in this article was implemented and integrated into the LLVM Compiler (version 2.9) as an alternate preallocation scheduler. The target machine in our experiments is x86 in both the 64-bit mode and the 32-bit mode. LLVM has a preallocation scheduler that is specifically designed to minimize register pressure. LLVM has two variants of this scheduler: a bottom-up register-pressure-reduction (BURR) scheduler and an ILP scheduler that tries to balance ILP and register pressure. By default, LLVM uses the ILP scheduler for the x86-64 target and the BURR scheduler for the x86-32 target. This choice is based on the assumption that on x86-32, the limited number of registers makes it critically important to schedule specifically for register pressure, while on x86-64, more registers are available and some degree of balance between ILP and register pressure may be achieved.

In both schedulers, however, the LLVM Compiler does not have precise latency information for the x86 target. Apparently, this is based on the assumption that for an out-of-order target machine, ILP is a secondary objective for the compiler. On x86-32, LLVM sets all latencies to unity, thus totally ignoring ILP and limiting the scheduling objective to minimizing register pressure. On x86-64, LLVM sets all latencies to unity except for the longer-latency instructions (such as the divide instruction). This enables some degree of ILP scheduling while still maintaining register pressure as the primary objective.

In this work, we report experimental results using both LLVM's *rough* latencies as well as the precise latency information that we have added to our combinatorial scheduler. In Sections 6.2 and 6.3, the combinatorial scheduler uses the same latencies as the LLVM scheduler, while in subsequent sections we study the impact of using precise latency information. Those results show that precise latency information can lead to significant performance gains on some benchmarks. Therefore, ILP should not be totally ignored on x86 in spite of the powerful out-of-order hardware.

The facts that x86 is an important target for LLVM and that LLVM has a register-pressure-reduction scheduler on x86 suggest that LLVM's preallocation scheduler is a good industry-strength reference for comparing our combinatorial scheduler with on x86.

The tests were run on a machine having two Intel Xeon E5540 processors running at 2.53 GHz with 24GB of memory. Each CPU has 8 threads (16 threads in total). All our tests, however, were compiled and executed using a single thread. Multithreading is beyond the scope of this work.

In all tests, LLVM was invoked with the –O3 optimization option as well as the following performance-tuning options: -march = core2, -mtune = core2. [1] With these options LLVM's default register allocator is used, which is a linear-scan allocator in LLVM 2.9. The operating system is Ubuntu 10.10 (the 64-bit version) for which LLVM's default target is the native x86-64 machine. To switch to the 32-bit mode on some tests, we have used the –m32 command-line option.

In running SPEC tests, random variations may occur. This randomness is due to the fact that a program's performance is affected by many factors, some of which, such as the state of the memory system, have some degree of randomness. In the experimental work reported in this article, special attention has been paid to minimizing the impact of these random factors. To this end, each test was run for a large number of iterations and the generated code was inspected to identify actual differences in instruction order and the amount of spilling. All the performance differences (gains or degradations)

[1] In our environment, LLVM 2.9 fails on 2 benchmarks in x86-64 mode and 5 benchmarks in x86-32 mode. These benchmarks were excluded in all the results reported in this article.

Table I. Hot Functions in FP2006 and Their Coverage

	BENCHMARK	# OF HOT FUNCS.	COVERAGE
1	Bwaves	3	96%
2	Milc	7	62%
3	Zeusmp	4	61%
4	Gromacs	2	75%
5	CactusADM	1	99%
6	Leslie	7	96%
7	Namd	10	71%
8	DealII	6	47%
9	Soplex	6	26%
10	Povray	5	38%
11	Calculix	3	75%
12	GemsFDTD	5	93%
13	Lbm	1	99%
14	Wrf	5	32%
15	Sphinx3	3	71%

that are reported in this article have been reproduced tens (and for some benchmarks hundreds) of times in the course of this experimental study that extended over two years. Furthermore, all the significant differences in performance have been linked to an actual difference in the generated code (mainly a difference in the amount of spill code in hot functions).

6.1. Hot Functions

To minimize the increase in compiler time, combinatorial scheduling was applied only to the hot functions in each benchmark. Hot function selection was based on the profiling data published by Weicker and Henning [2007].

Functions in which the program spends at least 5% of its execution time were selected, with a limit of 10 functions per benchmark. If a coverage of 90% of the benchmark's execution time was achieved with fewer functions, only those functions were selected. Table I shows the number of hot functions selected and the corresponding execution-time coverage for each benchmark in FP2006. Our selection criteria are intended to get a significant benefit with the minimum extra cost. As shown in the table, these criteria result in good coverage for most benchmarks and an almost complete coverage for benchmarks with peaky profiles such as Lbm and Cactus. Hot function selection is validated in Section 6.10 by applying the proposed algorithm to all functions.

6.2. Combinatorial Scheduling Statistics

Combinatorial scheduling was applied to the selected hot functions of all benchmarks in SPEC CPU2006 with a time limit of 10ms per instruction. So, a basic block with 100 instructions was given a time limit of one second. In this test, the combinatorial scheduler used the same latencies as the LLVM scheduler, and the Register Pressure Weight (RPW) was set to 1.

Since the combinatorial scheduler does not always converge into an optimal schedule within the time limit, the quality of the final schedule will depend on the quality of the initial heuristic schedule. Experimentally, however, we found the impact of this dependence to be quite limited. In most cases, the combinatorial scheduler gives a significant performance improvement regardless of the initial heuristic. For x86-64, using CP list scheduling as the initial heuristic gave slightly better results, while for x86-32 using LLVM's BURR heuristic gave slightly better results. To maximize the benefit,

Table II. Enumeration Statistics for CPU2006

	STAT	X86-64	X86-32
1	Total blocks processed	10805	10353
2	Blocks enumerated	347 (3%)	1603 (15%)
3	Blocks optimal and improved	274 (79%)	1136 (71%)
4	Blocks optimal and not improved	0 (0%)	197 (12%)
5	Blocks timed out and improved	65 (19%)	217 (14%)
6	Blocks timed out and not improved	8 (2%)	53 (3%)
7	Largest optimal block	569	292
8	Largest block improved	1145	1057
9	Smallest block timed out	71	38

Time limit = 10ms/instr. RPW = 1.

the better heuristic was used for each target. The cost of the heuristic schedule was then computed using Equation (1). If the cost is zero, the heuristic schedule is optimal; otherwise, the enumerator is invoked to search for a lower-cost schedule. During preallocation scheduling, the exact number of physical registers that are available for allocation is not known. So, LLVM makes some estimates, which tend to be pessimistic. For example, LLVM assumes that in x86-32 mode, only three 32-bit integer registers are available. These LLVM estimates are used in our cost computation. Combinatorial scheduling results are shown in Table II.

On x86-64, 10805 hot basic blocks were processed by the combinatorial scheduler. Only 3% of these basic blocks were passed to the enumerator. These are considered *hard* blocks. This percentage is misleading, because it may give the wrong impression that enumeration is unlikely to make a significant impact on performance. As shown in Section 6.8, the heuristic is more likely to fail on larger basic blocks, which generally have a higher impact on performance.

Row 3 shows that 79% of the hard blocks were optimally scheduled by the enumerator. For another 19% of the hard blocks, the enumerator did not find optimal schedules but still found schedules with lower costs than the heuristic schedules (Row 5). Summing the percentages in Rows 3 and 5 shows that 98% of the hard basic blocks were improved in cost by the enumerator.

Row 7 shows that the proposed algorithm was efficient enough to optimally schedule a basic block with 569 instructions. That basic block is from the *shell_* function in the *Bwaves* benchmark. For that basic block, the CP list schedule had a cost of 97, and the enumerator found a zero-cost schedule. The total combinatorial scheduling time for that basic block was 3.6 seconds. That large basic block, however, is one of many basic blocks in the *shell_* function, in which *Bwaves* spends only 6% of its execution time. Therefore, the significant improvement of that basic block's PERP did not have a measurable impact on the program's execution time.

In contrast, the largest improved basic block with 1145 instructions (Row 8) happened to be the basic block with the highest impact on the performance of *Cactus*. Thus, although that basic block was not solved optimally, improving its PERP had a significant positive impact on the program's execution time as listed in Table III(b) and explained in Section 6.8. Row 9 shows that all basic blocks with 70 instructions or less were solved optimally on x86-64.

Comparing the results for x86-32 with those for x86-64 shows that the scheduling problem on x86-32 is harder, because fewer registers are available for allocation on x86-32. Specifically, 15% of the basic blocks on x86-32 had heuristic schedules with nonzero costs and were then passed to the enumerator. The enumerator improved the schedules of 85% of these blocks (the sum of Rows 3 and 5). Row 4 shows that for 12%

Table III. SPEC CPU2006 Benchmark Scores

(a) INT2006 on x86-64

BENCHMARK	HEUR	COMB	% IMP
Perlbench	20.2	21.2	4.95%
Bzip2	13.6	13.9	2.16%
Gcc	19.9	19.5	−2.01%
Geo-mean (impacted)	**17.62**	**17.91**	**1.65%**
Geo-mean (overall)	**19.05**	**19.13**	**0.42%**

(b) FP2006 on x86-64

BENCHMARK	HEUR	COMB	% IMP
Milc	18.8	19.2	2.13%
Gromacs	11.2	11.5	2.68%
CactusADM	7.91	8.31	5.06%
Calculix	8.30	8.44	1.69%
Lbm	31.9	38.5	20.69%
Sphinx3	26.9	28.3	5.20%
Geo-mean (impacted)	**15.10**	**16.01**	**6.03%**
Geo-mean (overall)	**15.57**	**15.94**	**2.38%**

(c) INT2006 on x86-32 (PERP $\leq$ 40)

BENCHMARK	HEUR	COMB	% IMP
Perlbench	18.1	19.3	6.63%
Gobmk	16.8	17.0	1.19%
Hmmer	9.35	9.64	3.10%
Sjeng	17.8	17.5	−1.69%
Xalancbmk	20.9	21.2	1.44%
Geo-mean (impacted)	**16.03**	**16.36**	**2.06%**
Geo-mean (overall)	**16.98**	**17.14**	**0.94%**

(d) FP2006 on x86-32 (PERP $\leq$ 40)

BENCHMARK	HEUR	COMB	% IMP
Milc	15.3	17.3	13.07%
Zeusmp	13.5	13.7	1.48%
Leslie	8.97	9.33	4.01%
GemsFDTD	8.79	8.94	1.71%
Sphinx3	26.3	25.6	−2.66%
Geo-mean (impacted)	**13.38**	**13.83**	**3.36%**
Geo-mean (overall)	**15.22**	**15.43**	**1.38%**

Time limit = 10ms/instr. RPW = 1.

of the blocks passed to the enumerator, the enumerator proved that their nonzero-cost heuristic schedules were actually optimal.

6.3. Execution Times

Table III shows the SPEC CPU2006 scores using the proposed combinatorial scheduler (COMB) and LLVM's default heuristic scheduler (HEUR). The SPEC score represents the ratio between the execution time on a slow reference machine and the execution time on the test machine; so, a larger score indicates faster execution.

In this test, the combinatorial scheduler used the same latencies as the LLVM scheduler and the RPW was set to 1. The time limit is 10ms per instruction. As explained in Section 5, the proposed algorithm with the PERP cost function is expected to give the

best results on basic blocks that have optimal schedules with zero or near-zero PERP. Since on the x86-32 target, some basic blocks are unlikely to have such schedules, we have chosen not to apply the proposed algorithm to a basic block if its heuristic PERP exceeds 40. On the x86-64 target, however, the proposed algorithm was applied to *all* hot basic blocks.

Each benchmark was run 9 times using LLVM's heuristic scheduler and 9 times using the proposed combinatorial scheduler. The median score was taken for each scheduler and the difference between the two median scores was computed for each benchmark. If the difference for a given benchmark was negligibly small ($<\pm 1\%$), the difference was considered random variation, and the benchmark was considered to be *unimpacted* by the proposed algorithm. Table III shows the results for the *impacted* benchmarks.

The HEUR column in Table III shows the heuristic score, the COMB column shows the combinatorial score, and the %IMP column gives the percentage improvement of the combinatorial score relative to the heuristic score. The geometric mean in the next-to-last row of each table is computed across the impacted benchmarks, while the geometric mean in the last row of each table is computed across the entire benchmark suite (INT2006 or FP2006). In computing the overall geometric mean across the entire suite, the difference for the unimpacted benchmarks, which are not listed in the table, was set to zero.

Inspecting the improvements in these tables shows that the combinatorial algorithm makes its most significant improvement on FP2006 in the x86-64 mode with a geometric-mean improvement of 6.03% across impacted benchmarks and 2.38% across the entire FP2006 benchmark suite. An improvement of 21% is obtained on Lbm. The FP benchmarks benefit from combinatorial scheduling more than the INT benchmarks, because FP benchmarks have larger basic blocks with higher degrees of ILP, higher register pressure values, and higher impacts on performance. The heuristic scheduler is more likely to fail on such blocks. More improvement is seen on X86-64 than on x86-32, because the availability of more physical registers on x86-64 makes it more likely for a basic block to have a schedule with zero or near-zero PERP. As explained in Section 5, such basic blocks are more likely to benefit from the proposed algorithm with the PERP cost function. The results in most subsequent sections will focus on the scheduling of FP2006 on x86-64.

The cost improvements reported in the previous section did not always translate into actual improvements of the execution time for the following reasons.

—Many hot functions have large numbers of basic blocks, and many of these blocks have low profiles within those functions. Improvements of basic blocks with relatively low profiles (such as the large basic block in *Bwaves* mentioned in Section 6.2) may not result in measurable improvements in execution time.
—The cost improvements reported in Section 6.2 are based on LLVM's pessimistic estimates of available physical registers during scheduling. During register allocation, more physical registers may be available. Thus, a block with a nonzero PERP may be allocated without spilling.
—LLVM performs other optimizations, such as Common Subexpression Elimination (CSE) and Loop-Invariant Code Motion (LICM) between instruction scheduling and register allocation. Those optimizations may affect register pressure.
—The cost function and locality limitations discussed in Section 5 weaken the correlation between the scheduling objective and the actual execution time.

6.4. Precise Latency Information

As mentioned earlier, the LLVM Compiler does not use precise latency information for the x86 target. The results reported in the previous sections were based on the

Table IV. Combinatorial Scheduling of CPU2006 with Precise Latency Information on x86-64

	STAT	LLVM LATENCY	PRECISE LATENCY
1	Total blocks processed	10805	10805
2	Blocks enumerated	347 (3%)	396 (4%)
3	Blocks optimal and improved	274 (79%)	301 (76%)
4	Blocks optimal and not improved	0 (0%)	3 (1%)
5	Blocks timed out and improved	65 (19%)	78 (20%)
6	Blocks timed out and not improved	8 (2%)	14 (4%)
7	Largest optimal block	569	569
8	Largest block improved	1145	1145
9	Smallest block timed out	71	53

Time limit = 10ms/instr. RPW = 1.

latencies used by the LLVM Compiler. In this section we study the impact of using precise latency information. The latencies used in our experiments are based on Fog's work [Fog 2012].[2] The results in this section were generated using an RPW of 1. Results using different RPW values are presented in the next section.

Table IV shows combinatorial scheduling statistic for CPU2006 using both LLVM latencies and precise latencies. These numbers show that the scheduling problem becomes harder when precise latencies are used, but the difference is not enormous. When precise latencies are used, more basic blocks are passed to the enumerator (4% versus 3%) and a smaller percentage of these blocks (76% versus 79%) are solved to optimality within the time limit of 10ms per instruction. However, the vast majority of the basic blocks are improved by the combinatorial scheduler in both cases.

Table V shows the impact of using precise latency information on the overall performance of FP2006. The numbers in this table are percentage performance improvements relative to LLVM's default heuristic scheduler. In addition to the six benchmarks reported in Table III(b), four more FP2006 benchmarks (Bwaves, Leslie, Namd, and Gems) are impacted by preallocation scheduling when precise latencies are used.

The third column in the table shows the percentage speedup using precise latencies with a time limit of 10ms per instruction, while the fourth column shows the percentage speedup using precise latencies with a time limit of 80ms per instruction. Only two benchmarks (Gromacs and Calculix) benefit from the extra compile time. This shows that most of the benefit from the proposed combinatorial algorithm is achieved with a relatively small time limit. No further increase in performance has been detected with time limits above 80ms per instruction. A closer look at the impact of increasing the time limit is presented in Section 6.9.

All impacted benchmarks are improved when precise latencies are used except for Cactus and Gems, which incur small performance degradations relative to the LLVM scheduler. The causes of these degradations are discussed in Section 6.8. As shown in the next section, however, these degradations may be eliminated by using different RPW values. The most interesting result in Table V is perhaps the double-digit percentage improvement on Gromacs when precise latencies are used. Examining this benchmark's hot code shows that it has a large basic block with 315 instructions that has many long-latency instructions. This result suggests that the out-of-order processor with its 96-entry reorder buffer may not be fully exploiting the available ILP in this benchmark's hot code.

[2]Latencies for the major and most frequently used instructions were entered; latencies for all other instructions were set to unity.

Table V. Percentage Speedup Relative to the LLVM Heuristic Using Two Different Levels of Latency Precision

BENCHMARK	LLVM LATENCIES 10 MS/INSTR	PRECISE LATENCIES 10 MS/INSTR	PRECISE LATENCIES 80 MS/INSTR
Bwaves	0.00	1.97	1.97
Milc	2.13	3.78	3.78
Gromacs	2.68	0.89	10.71
CactusADM	5.06	−2.44	−2.44
Leslie	0.00	−0.91	−0.91
Namd	0.00	1.25	1.25
Calculix	1.69	2.41	3.13
Gems	0.00	−1.89	−1.89
Lbm	20.69	22.26	22.26
Sphinx3	5.20	4.83	4.83
Geo-mean (impacted)	**3.60%**	**3.02%**	**4.05%**
Geo-mean (overall)	**2.38%**	**2.00%**	**2.68%**

Target: x86-64. RPW = 1.

Table VI. Percentage Speedup on FP2006 Relative to the LLVM Heuristic on x86-64 for Different RPW Values

BENCHMARK	RPW=0	RPW=1	RPW=2	RPW=3	LARGE PRW
Bwaves	0.00	1.97	1.97	1.97	0.00
Milc	3.78	3.78	3.78	3.78	2.13
Gromacs	−3.57	10.71	11.61	8.93	2.68
CactusADM	−7.19	−2.44	−0.90	0.00	5.06
Leslie	−1.82	−0.91	−0.91	−0.91	0.00
Namd	−1.88	1.25	1.25	1.25	0.00
Calculix	3.61	3.13	2.29	2.29	1.69
Gems	−3.77	−1.89	−1.89	−1.89	0.00
Lbm	17.55	22.26	22.26	22.26	20.69
Sphinx3	4.09	4.83	4.83	4.83	5.20
Geo-mean (impacted)	**0.88%**	**4.05%**	**4.21%**	**4.05%**	**3.60%**
Geo-mean (overall)	**0.59%**	**2.68%**	**2.79%**	**2.68%**	**2.38%**

Target: x86-64. Time limit = 10ms/instr (except for Gromacs and Calculix).

6.5. Register Pressure Weight

In this section we study the impact of the Register Pressure Weight (RPW). Table VI shows the percentage performance improvements on FP2006 by the proposed combinatorial algorithm relative to the LLVM heuristic scheduler using different RPW values. The time limit is 10ms per instruction, except for Gromacs and Calculix. The time limit was set to 80ms per instruction for Gromacs and to 50ms per instruction for Calculix, so that the best possible improvement is achieved for these two benchmarks.

The first observation is that as long as register pressure is taken into account (RPW $\geq$ 1), the proposed combinatorial algorithm gives significant performance improvements relative to the LLVM scheduler without causing any substantial performance degradation. The maximum degradation with RPW $\geq$ 1 is 2.44% (seen on Cactus with RPW = 1). Substantial degradations are only seen in the column with RPW = 0, where register pressure is totally ignored (pure ILP scheduling). The most significant degradation with RPW = 0 is seen on Cactus (7.19%), which has particularly high register pressure as detailed in Section 6.8.

The results in the table show that the best geometric-mean improvement is obtained with a RPW = 2. However, using other moderate RPW values (1 or 3) gives comparably good results. It should be noted here that using an RPW of 5 gave the same results as using an RPW of 3.

The interesting observation is that using precise latency information and then setting the RPW to a large value is, from a practical point of view, an inefficient way of emphasizing register pressure and deemphasizing ILP. When a basic block has many long-latency instructions, the schedule produced by the proposed combinatorial algorithm will overlap the execution of these long-latency instructions, if possible, to reduce the schedule length. Since more overlapping of instructions results in more overlapping of live ranges, this will tend to increase the register pressure unless the RPW is extremely large. Setting the RPW to a very large value, however, will slow the combinatorial search, because the combinatorial scheduler will first search for a short schedule with low register pressure, then if no such a schedule is found, it will search for successively longer schedules with lower register pressures. Longer schedules with lower register pressures will involve less overlapping of long-latency instructions (more sequential and less parallel execution). From a practical perspective, such longer schedules with less overlap can be found much more efficiently by limiting the number of long latencies in the DDG. In the limit, setting all latency values to unity effectively eliminates the ILP component of the cost function and schedules for the sole objective of minimizing register pressure. The results in the last column of Table VI were obtained using LLVM's rough latency values (which are mostly unity) as an efficient pragmatic way of implementing a very large register pressure weight. Note that the results in that column are the same as the results in Table III(b).

Generally speaking, using moderate RPWs (1, 2, or 3) to balance ILP and register pressure gives better performance results than using extremely low or extremely high RPWs. Only two benchmarks (Cactus and Gems) have better performance when a very large RPW is used. Examining the hot code in these two benchmarks shows that they both have large basic blocks with high register pressures and many long-latency instructions. The overlapping that results from using precise latency information produces schedules with high register pressures for these two benchmarks. A more detailed analysis of these two benchmarks is given in Section 6.8.

At the other end of the spectrum, setting the RPW to zero (pure ILP scheduling) causes many performance degradations. The most significant degradations are seen on Cactus (7.19%), Gems (3.77%), and Gromacs (3.57%). This result is not surprising, because these benchmarks have very high register pressure in their hot code. Only one benchmark, namely Calculix, has slightly better performance with pure ILP scheduling. Examining the code in this benchmark's hottest function (e_c3d_ with a profile of 54%) shows that this function has many CPU-intensive basic blocks whose register pressures are not heavily dependent on the schedule. Therefore, the performance differences reported in the table for Calculix are likely to be due to ILP.

It should be noted here that although the performance differences reported for Calculix are relatively small, they are highly deterministic and were consistently reproduced in all iterations. Unlike some other benchmarks, no random variation has been observed on Calculix throughout the experimental study reported in this article. This high degree of determinism is attributed to the fact that this benchmark's hot code is CPU-bound and has negligible memory sensitivity.

The results in Table VI show that different benchmarks reach their best performance with different RPWs. For example, consistently better performance is achieved on Cactus as the RPW is increased, which indicates that this benchmark's performance is highly dependent on the amount of spill code. As detailed in Section 6.8, Cactus has

Table VII. Percentage Speedup on FP2006 Relative to the LLVM Heuristic Using an Automatically Tuned RPW

BENCHMARK	RPW=2
Bwaves	1.32
Milc	3.78
Gromacs	11.61
CactusADM	5.91
Leslie	−0.91
Namd	1.25
Calculix	2.29
Gems	−1.89
Lbm	22.26
Sphinx3	4.83
Geo-mean (impacted)	**4.84%**
Geo-mean (overall)	**3.20%**

Target: x86-64. Time limit = 10ms/instr (except for Gromacs and Calculix).

hundreds of spills in its hottest function. The Gromacs benchmark, on the other hand, exhibits a different pattern. The best performance for this benchmark is achieved with an RPW of 2, which indicates that balancing register pressure and ILP is critically important on this benchmark.

In conclusion, the register pressure weight is a parameter that can be used to tune the performance of a given program. This can be done either manually or automatically. In automatic tuning, various heuristics may be used to select the RPW for a given basic block on a given target processor. We did not invest much time experimenting with automatic tuning, but one tuning heuristic that gave good results was using LLVM latencies for very large basic blocks and precise latencies for all other blocks. The rationale behind this choice is that using precise latency information makes the scheduling problem harder, and thus more compile time will be needed to achieve a good balance between ILP and register pressure. Since our experimental results show that this balance on x86 is likely to favor low register pressure, a pragmatic way of achieving this in less compile time is using less precise latency information. Table VII shows the percentage speedups relative to the LLVM scheduler using LLVM latencies for blocks with more than 500 instructions and precise latencies for all other blocks. With precise latencies, an RPW of 2 was used, while with LLVM latencies an RPW of 1 was used. As shown in the table, this tuning scheme gives a geometric-mean improvement of 3.2% across FP2006.

6.6. Compile Times

Table VIII shows the total compile times for all FP2006 benchmarks on x86-64. Total compile times are shown for the LLVM heuristic scheduler (HEUR), the combinatorial scheduler with LLVM latencies (COMB LLVM LAT), and the combinatorial scheduler with precise latencies (COMB PREC LAT). With LLVM latencies, the time limit is 10ms per instruction for all benchmarks. With precise latencies, the time limit is 10ms per instruction for all benchmarks except for Gromacs (80ms/instr) and Caluculix (50ms/instr). Each of the last two columns also shows the percentage increase in compile time relative to the base compile time using the LLVM heuristic.

The compile times in this table show the practicality of the proposed algorithm compared to previous work on combinatorial scheduling [Barany and Krall 2013; Govindarajan et al. 2003; Kessler 1998; Malik 2008; Winkel 2007].

Table VIII. Compile Times of FP2006 on x86-64

BENCHMARK	HEUR (S)	COMB LLVM LAT (S)	COMB PREC. LAT (S)
Bwaves	3	15 (400%)	15 (400%)
Milc	6	7 (17%)	7 (17%)
Zeusmp	18	45 (150%)	47 (161%)
Gromacs	24	27 (13%)	48 (100%)
CactusADM	32	59 (84%)	59 (84%)
Leslie	5	10 (100%)	10 (100%)
Namd	9	44 (389%)	50 (456%)
DealII	128	129 (1%)	129 (1%)
Soplex	30	30 (0%)	30 (0%)
Povray	33	33 (0%)	33 (0%)
Calculix	45	48 (7%)	55 (22%)
GemsFDTD	13	21 (62%)	58 (346%)
Lbm	2	5 (150%)	5 (150%)
Wrf	162	216 (33%)	218 (35%)
Sphinx3	7	7 (0%)	7 (0%)
Total	**517**	**696 (35%)**	**771 (49%)**

Time limit = 10ms/instr. RPW = 1.

For many benchmarks, using the proposed algorithm causes minimal increase in compile time. All compile times are within a reasonably practical range. Unlike previous work on combinatorial scheduling, there are no long compile times of many hours. With combinatorial scheduling applied only to the hot functions, the entire FP2006 benchmark suite is compiled in 11.6 minutes (35% increase) using LLVM's latencies and in 12.9 minutes (49% increase) using precise latencies. Although these are significant increases in compile time, the compile times with combinatorial scheduling are still quite practical. It is noted here that the compilation process of large applications is highly parallelizable. So, the increase in compile time can possibly be absorbed by utilizing a multithreaded processor to compile multiple source files in parallel. This feature is already offered by existing build utilities such as *gmake*.

Comparing the third and fourth columns in Table VIII shows that using precise latency values results in a significant increase in compile time on some benchmarks such as Gems and Namd, which have many long-latency instructions in their hot code. For most benchmarks, however, using precise latencies does not lead to significant increases in compile time relative to using LLVM latencies.

6.7. Pruning Techniques

In this section we study the effectiveness of the different pruning techniques described in this article. The effectiveness of each pruning technique is evaluated by disabling that technique and measuring the resulting increase in the total compile time for CPU2006 (second column in Table IX) and the decrease in the number of blocks solved optimally (third column in Table IX). These results show that the most effective pruning technique is the register pressure cost test that checks at each tree node if the current register pressure cost is less than the best known cost. Disabling this technique increases the total compile time by 79% and decreases the number of basic blocks solved optimally by 89%.

The results in the table also show that history-based domination is an effective technique with an impact of 12% on total compile time and 11% on optimally scheduled blocks. Relaxed scheduling has a relatively low impact on the combinatorial scheduler's

Table IX. Pruning Techniques

PRUNING TECHNIQUE	INCREASE IN COMPILE TIME	DECREASE IN OPTIMAL BLOCKS
Relaxed Scheduling	3%	4%
Register Pressure Cost Test	79%	89%
History-Based Domination	12%	11%

Target = x86-64. Time limit = 10ms/instr. RPW = 1.

speed. This is attributed to the tightness of the schedule-length lower bounds that are computed using Langevin and Cerny's algorithm [1996] before enumeration. In all the nontrivial test cases that we have examined, many feasible schedules do exist at the schedule-length lower bound, and the enumerator spends most of its time searching for a feasible schedule with minimum register pressure at that length. The combination of a tight schedule-length lower bound and a register-pressure-cost pruning technique distinguishes the proposed algorithm from previous work.

6.8. Case Studies

In this section we shed more light on four benchmarks that were improved by the proposed algorithm and two benchmarks that incurred performance degradations. The analyses that follow are based on the profiling information published by Weicker and Henning [2007].

Milc on x86-32. The maximum improvement on x86-32 occurred on Milc**.** For this benchmark, combinatorial scheduling was applied to the top 7 hot functions, which cover 62% of the program's profile. These 7 functions have 11 basic blocks. LLVM's BURR heuristic scheduler produces zero-cost schedules for 7 basic blocks. The remaining 4 basic blocks are passed to the enumerator to search for an optimal schedule. These 4 basic blocks happened to be the dominant basic blocks in the top four hot functions that cover 47% of the program's profile.

The top hot function *mult_su3_na* in Milc has two small basic blocks with 3 and 11 instructions and a large basic block with 117 instructions. Therefore, the function's performance is likely to be dominated by the code quality in the large basic block. The heuristic schedule for this block has a schedule length of 117 (which is optimal) and a PERP of 11. With this PERP, the register allocator spills 18 virtual registers.

The enumerator incrementally improves this heuristic schedule by first finding a schedule with a PERP of 10, then continuing until a schedule with a PERP of 2 is found. The latter schedule is proven optimal by completing the exhaustive search. Using the pruning techniques described in the article, the search takes 230ms. With this reduced PERP, LLVM's register allocator does not spill any virtual registers. Similar eliminations of 18 spills per function happen in the other three hot functions (which have similar structure), thus eliminating a total of 72 hot spills and improving the benchmark's execution time by 13%. Enabling combinatorial scheduling on Milc's top 7 hot functions increases the benchmark's total compile time by only one second (from 6 to 7 seconds). Milc then gives a good example of effective combinatorial scheduling with minimal increase in compile time.

Lbm on x86-64. Lbm has a very peaky profile with 99% of its execution time spent in one function. This hot function has 6 basic blocks: one large basic block with 274 instructions and five smaller basic blocks with sizes ranging from 3 to 44 instructions. The small basic blocks are found to have optimal heuristic schedules, while the large basic block has a heuristic schedule length of 276 (2 cycles above the lower bound of 274) and a PERP of 18. This schedule causes the register allocator to spill 12 virtual registers. As

it turns out, spilling 12 virtual registers in this hot basic block considerably degrades Lbm's performance.

When combinatorial scheduling is applied to this large basic block using LLVM latencies, the CP list schedule has a schedule length of 274 and a PERP of 5. The enumerator finds an improved schedule with length 274 and a PERP of 4, then it times out. The improved schedule results in only 2 spills, a significant reduction relative to LLVM's heuristic. That gives an improvement of 21% in the program's execution time as listed in Table III(b). The improved schedule with a PERP of 4 is found in 1.8 seconds. With a time limit of 10ms per instruction on x86-64, the time limit for this block is 2.74 seconds.

When precise latencies are used, the best schedule has the same PERP but slightly improved ILP, which leads to a slightly larger speedup of 22%.

Cactus on x86-64. Cactus is another peaky benchmark that spends 99.9% of its execution time in one function. Unlike Milc and Lbm, Cactus's hot function has multiple large basic blocks with very high register pressure values, and it is unlikely that schedules with low PERPs exist. The function has 64 basic blocks. LLVM's heuristic schedules for these basic blocks cause the register allocator to spill 662 virtual registers in the whole function.

When combinatorial scheduling is applied using LLVM latencies, the CP heuristic produces zero-cost schedules for 50 blocks. When the remaining 14 basic blocks are passed to the enumerator with a time limit of 10ms per instruction, 10 basic blocks are solved optimally and 4 large basic blocks time out. For all 14 basic blocks, the enumerator finds schedules with improved costs relative to the CP heuristic. The sizes of the basic blocks that time out are 256, 459, 713, and 1145 instructions. Based on our experimentation, the largest basic block with 1145 instructions has the highest impact on performance. The CP heuristic produces a schedule with length 1145 (optimal) and a PERP of 133. The enumerator first finds a feasible schedule with length 1145 and a PERP of 119, then it incrementally improves this until it finds a schedule with length 1145 and a PERP of 104 before it times out. The time limit for this block is 11.45 seconds.

The best schedule found by the enumerator within this time limit results in reducing the function's spill count from 662 to 621. As shown in Table III(b), this spill code reduction improves the program's execution time by 5%. Longer time limits were attempted for Cactus, and the enumerator exhibited a very slow convergence pattern that only resulted in a slight improvement in the spill count without leading to a measurable improvement in execution time. The optimal schedule for Cactus's largest hot block remains unknown at this point. Cactus is then an example of a benchmark that is improved by the combinatorial algorithm even though the algorithm does not converge into an optimal schedule.

When precise latencies are used with an RPW of 1, the enumerator times out with a best schedule of length 1145 and a PERP of 107. Although this PERP is almost equal to the PERP of the best schedule found using LLVM latencies, the resulting spill count in this case is 676, which is larger than the spill count obtained using the LLVM scheduler. This causes a degradation of 2.44% relative to the LLVM scheduler. The increase in the spill count in this case is attributed to the increased overlapping of long-latency instructions to produce a schedule with length 1145. The increased level of overlap among live ranges that results from this parallelism is not fully captured by the PERP cost function. To get a better indication about the degree of overlap, code was added to measure the ERP at all points in each schedule and then print for each basic block the number of points at which each scheduler (combinatorial or heuristic) has a higher ERP. The results showed that with LLVM latencies, the combinatorial schedule has a lower ERP at 829 points and a higher ERP at 172 points, while with precise latencies, the

combinatorial schedule has a lower ERP at 677 points and a higher register pressure at 309. This experimental result is consistent with the hypothesis discussed in Section 5.1 that the PERP cost function has a strong but not perfect correlation with the amount of spill code. The correlation is weaker when the basic block is larger and when there are more long-latency instructions. We should note here that in all other enumerated blocks in Cactus, the combinatorial schedules were found to have lower ERPs than the heuristic schedules at *every* point in the schedule. This shows that for most blocks, minimizing the PERP gives the desired results.

When an RPW of 2 is used, the best schedule has a PERP of 105. This schedule has a lower register pressure at 757 points and a higher register pressure at 240 points, which leads to a spill count of 669 and an execution-time degradation of 0.9%.

Gromacs on x86-64. Gromacs is a benchmark that shows the importance of ILP scheduling even when the target processor has out-of-order execution. Gromacs's performance is heavily dependent on the hot function *inl1130_*, in which the program spends 66% of its execution time. This function has 18 basic blocks. Only 3 basic blocks are passed to the enumerator, while the other 15 (mostly small) basic blocks have zero-cost heuristic schedules. Two of the three enumerated blocks (with sizes 65 and 75 instructions) are solved optimally by the enumerator in less than 1ms each, with small improvements (2 and 3) in PERP. These small improvements do not seem to have a significant impact on performance. Experimentation shows that Gromacs's performance is highly impacted by the schedule in the third and largest enumerated block, which has 315 instructions. The analysis that follows will focus on this basic block.

When LLVM latencies are used, 95% of the edges in this block's DDG are assigned unit latencies. The CP list schedule for this block has a length of 315 and a PERP of 44. The enumerator incrementally improves this until it times out with a feasible schedule with length 315 and a PERP of 33. The resulting schedule leads to slightly more spills compared to the LLVM schedule (89 compared to 86), but the code runs 2.68% faster. This speedup is believed to be caused by improved ILP scheduling.

When precise latencies are used, instructions have a wider distribution of latencies including latencies of 1, 2, 3, 4, and 7. Only 39% of the DDG edges are now assigned unit latencies. The combinatorial scheduler's behavior in this case is similar to its behavior with LLVM latencies in terms of PERP. With an RPW of 1, the enumerator times out with a best schedule of length 315 and PERP 32, which is about the same as that found using LLVM latencies and leads to approximately the same spill count (85 spills). With this schedule, however, the benchmark runs 11% faster relative to the base LLVM compiler. Examining the combinatorial schedule shows much better spacing of long-latency instructions. This suggests that the performance improvement in this case is due to improved ILP scheduling in this large basic block, whose size is much larger than the target processor's reorder buffer.

When an RPW of 2 is used, a slightly better balance is achieved between ILP and register pressure. The enumerator times out in this case with a schedule length of 318 and a PERP of 31. This results in 84 spills and leads to a speedup of 12%.

Gems on x86-64. Gems has five hot functions with profiles ranging from 22% to 14%. These hot functions have a total of 305 basic blocks. Given this relatively large number of blocks, most of these basic blocks will not have a significant impact on overall performance. When LLVM latencies are used, 83 basic blocks (27%) are passed to the enumerator. The enumerator optimally schedules all but two basic blocks and improves the PERP of all blocks. However, most of these PERP improvements are relatively small (2 or 3). Combining this with the fact that most basic blocks have relatively low profiles explains why these PERP reductions do not result in measurable runtime improvements.

When precise latencies are used, 103 basic blocks (34%) are passed to the enumerator. The increase is due to the fact that with more long-latency instructions, heuristic schedules are less likely to have zero costs. Interestingly, the combinatorial scheduler degrades performance by about 2% in this case as shown in the middle three columns in Table VI. Examining the spill counts reported by the register allocator shows that combinatorial scheduling increases the amount of spill code in three hot functions. To understand the causes of this negative result, the ERP at *all* points in each schedule was examined as in the Cactus case. The results showed that in 6 out of 103 enumerated blocks in Gems, the combinatorial schedule has higher ERP than the heuristic schedule at multiple points. The extreme case is seen in two similar basic blocks with 91 instructions each, where the combinatorial scheduler has a higher ERP than the heuristic scheduler at 39 out of 91 points in the schedule.

When similar measurements were made using LLVM latencies, only 1 out of 83 enumerated blocks had a few points at which the combinatorial ERP is higher than the heuristic ERP. This explains the performance degradation that is seen when precise latencies are used. As explained before, using precise latencies tends to produce schedules with more overlapping of parallel instructions, thus making it more likely to increase the register pressure at some points in the schedule. Similar to the Cactus case, these experimental results confirm the hypothesis that the PERP exhibits a strong but not perfect correlation with the amount of spill code. Note that even in the precise-latency case, the combinatorial ERP was higher than the heuristic ERP at *all* points in 97 out of 103 blocks. Viewed positively, this result shows that in 94% of the Gems blocks, minimizing the PERP minimizes the ERP at *all* points in the schedule. Statistically, this is a strong though not perfect correlation.

The rarity of the negative numbers in Table VI (excluding the zero-RPW column) suggests that the correlation between the PERP and the register pressure at *all* points in the schedule is very strong in most hot basic blocks.

GCC on x86-64. Table III(a) shows that the combinatorial algorithm slows GCC by 2% relative to the LLVM scheduler. To explain this degradation, a closer look was taken at GCC's hot code. GCC has a flat profile. In our tests, combinatorial scheduling was applied to the top four hot functions whose profiles are 5% or greater for any invocation, as reported by Weicker and Henning [2007].[3] These hot functions have a total of 88 small basic blocks ranging in size from 3 to 21 instructions. All basic blocks have zero-cost heuristic schedules, and therefore no basic block is passed to the enumerator for this benchmark. For three out of four hot functions, the spill count is zero, while for the fourth hot function, 5 spills are reported. Identical spill counts for all four functions are reported when the LLVM scheduler is used. Therefore, the performance degradation has nothing to do with spill code. Like many integer benchmarks, GCC does not have high register pressure in its hot code. Poor ILP scheduling within the basic block (local scheduling) is unlikely to be the cause of the degradation, because such small basic blocks do not have enough ILP. This leads to the conclusion that the degradation must be caused by a factor that is not covered by the scheduler's objective. This factor could be global scheduling across basic blocks or memory-system performance. As mentioned in the Introduction, the proposed algorithm, like any other compiler optimization, addresses only a subset of the factors that affect performance. The fact that such performance degradations are both rare and small in size suggests that the proposed algorithm performs very well in practice.

[3]The library function *memset* had to be excluded.

Table X. Impact of Increasing the Time Limit

	STAT	10 MS/INSTR	40 MS/INSTR	80 MS/INSTR	160 MS/INSTR
1	Total blocks processed	10805	10805	10805	10805
2	Blocks enumerated	396 (4%)	396 (4%)	396 (4%)	396 (4%)
3	Optimal and improved	301 (76%)	310 (78%)	317 (80%)	319 (81%)
4	Optimal and not improved	3 (1%)	3 (1%)	3 (1%)	3 (1%)
5	Timed out and improved	78 (20%)	69 (17%)	62 (16%)	60 (15%)
6	Timed out and not improved	14 (4%)	14 (4%)	14 (4%)	14 (4%)
7	Largest optimal block	569	569	569	569
8	Largest block improved	1145	1145	1145	1145
9	Smallest block timed out	53	79	79	79

Target: x86-64. RPW = 1.

6.9. A Closer Look at Timeouts

In this section we study the impact of the time limit. We start by taking a look at the smallest problem that times out. According to Table IV, when precise latencies are used with an RPW of 1, the size of the smallest problem that times out is 53 instructions. With a time limit of 10ms/instr, this problem is given a time limit of 530ms. To see how hard this problem is, the test was run with no time limit for this block, and the combinatorial algorithm solved the problem optimally in 1.7s, which indicates that this problem is not indeed a hard problem.

When the time limit was increased to 40ms/instr, the smallest problem that timed out had 79 instructions. That problem was examined more closely. First, it was noticed that the problem has a wide distribution of latencies ranging from 1 to 6. To determine if the hardness of this problem is caused by this wide range of latencies, the test was run with all latencies set to unity. The problem still timed out. The heuristic schedule for this block had a length of 79 (at the lower bound) and a PERP of 6. The enumerator finds an improved schedule with length 79 and a PERP of 5 in less than one millisecond. It then times out while searching for a schedule with PERP less than 5. To measure the time needed to solve this problem, the test was run with no time limit and the problem was solved optimally in 615 seconds (about 10 minutes) with length 79 and a PERP 5. So, the enumerator actually found the optimal schedule in less than one millisecond, and then it spent 10 minutes of exhaustive search to prove that this schedule is indeed optimal. Obviously, such a proof of optimality has very little practical value.

A similar behavior has been observed on many other test cases. In fact, larger time limits have been tried for the entire benchmark suite (CPU2006), and for most blocks, the outcome was either a slight improvement of the PERP (by a few units) or a proof of optimality with no improvement at all. Neither of these outcomes led to an actual improvement in performance. The interesting observation when studying these test cases was the *decelerated convergence* into an optimal schedule. For a typical hard block with hundreds of instructions, the enumerator quickly finds many improved schedules within a few seconds or even a fraction of a second, and then it takes minutes or even hours to find the next better schedule. If the block is hot enough, the best schedule found within the first few seconds will lead to a measurable performance improvement.

Table X characterizes the impact of increasing the time limit on CPU2006. Increasing the time limit only moves a few basic blocks from the timedout-and-improved category to the optimal-and-improved category. None of these time limit increases has resulted in a *measurable* impact on runtime performance. This shows that, from a practical point of view, all the benefit from the proposed algorithm is obtained with a relatively small time limit.

Table XI. Statistic for Combinatorial Scheduling of All Functions in FP2006

	STAT	VALUE
1	Total blocks processed	379744
2	Blocks enumerated	9456 (2.5%)
3	Blocks optimal and improved	7645 (80.8%)
4	Blocks optimal and not improved	142 (1.5%)
5	Blocks timed out and improved	1371 (14.5%)
6	Blocks timed out and not improved	298 (3.2%)
7	Largest optimal block	664
8	Largest block improved	5024
9	Smallest block timed out	32
10	Avg. block size	15
11	Avg. enumerated block size	99
12	Avg. optimal solution time	36 ms

Target: x86-64. Time limit = 10ms/instr. RPW = 1.

6.10. Combinatorial Scheduling of all Functions

The data presented in Sections 6.4 and 6.6 show that when combinatorial scheduling with precise latencies and an RPW of 1 is applied to the hot functions in FP2006 on x86-64, it gives a geometric-mean performance improvement of 2.7% at the cost of increasing the total compile time by only 4.3 minutes. From a practical point of view, this *selective* application of combinatorial scheduling is the recommended usage model in a production compiler. This can be implemented by using profile feedback and applying combinatorial scheduling only to those basic blocks whose execution frequencies are above a certain threshold. More generally, the time limit per basic block will be a function of the basic block's execution frequency as well as the basic block size.

However, to test the robustness of the proposed algorithm and collect statistics for a large dataset, we conducted an experiment in which combinatorial scheduling was applied to *all* functions in FP2006 on x86-64. The outcome of that experiment was that the total compile time increased from 12.9 minutes to one hour and 15 minutes without producing any additional significant improvement in execution time.[4] This result validates our hot function selection. As shown in Table I, the selected hot functions provide good profile coverage for most FP2006 benchmarks. In Table I there are only four benchmarks that are lightly covered due to their flat profiles: dealII, soplex, povray, and wrf. Examining the spilling information for these four benchmarks shows that the only benchmark with a significant amount of spilling is wrf. Apparently, LLVM's heuristic scheduler produces good schedules for this particular benchmark. Table XI shows the statistics that were collected during the all-function experiment.

Combinatorial scheduling was applied to a total of 379744 basic blocks. 97.5% of these blocks had zero-cost heuristic schedules, and the remaining 2.5% were passed to the enumerator. Row 10 shows that the average block size in FP2006 is 15 instructions, while Row 11 shows that the enumerated blocks were fairly large blocks with an average size of 99 instructions. These blocks are likely to have higher impacts on performance. The enumerator optimally scheduled 80.8% of these blocks (Row 3) and improved the schedules of another 14.5% (Row 5). Row 7 shows that the enumerator optimally solved blocks as large as 664 instructions. Row 8 shows that the enumerator improved the schedules of blocks with thousands of instructions. The average combinatorial scheduling time across the blocks that did not time out is 36ms per block.

[4]In this all-function experiment a runtime error occurred on the Cactus benchmark.

7. CONCLUSIONS AND FUTURE WORK

This article presents a combinatorial algorithm for balancing ILP and register pressure in preallocation scheduling. The algorithm uses branch-and-bound enumeration to minimize a cost function that is a linear combination of schedule length and register pressure. If applied selectively to the hot spots, this algorithm significantly improves the performance of many programs with a reasonable increase in compile time. Most of the benefit from this algorithm can be obtained using relatively low time limits that keep the total compile time within a practically acceptable range.

The results of this article show that greedy heuristic approaches to preallocation scheduling may result in significant performance degradations due to either increasing register pressure or missing ILP opportunities.

Future work will attack the more complex problem of integrated instruction scheduling and register allocation to achieve a stronger correlation between the scheduling objective and the amount of spill code.

ACKNOWLEDGMENTS

The authors thank the computer science (CS) department and the deanship of academic research at PSUT for providing the hardware and software that were used in the experiments. Special thanks go to Mohammad Abusaad, the system administrator at PSUT, for the technical support that he provided to us on the test machine. We also thank Andrew Trick, the instruction scheduling code owner in LLVM, for answering our questions about the LLVM scheduler. Finally, we thank the anonymous reviewers for their constructive comments. Thanks to the reviewers' insightful comments, the article now has an extensive experimental evaluation that we hope will benefit researchers and practitioners in the field.

REFERENCES

Barany, G. 2011. Register reuse scheduling. In *Proceedings of the 9th Workshop on Optimizations for DSP and Embedded Systems (ODES'11).*

Barany, G. and Krall, A. 2013. Optimal and heuristic global code motion for minimal spilling. In *Proceedings of the International Conference on Compiler Construction.*

Berson, D., Gupta, R., and Soffa, M. 1993. URSA: A unified resource allocator for registers and functional units in VLIW architectures. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism.* 243–254.

Cooper, K. and Torczon, L. 2004. *Engineering a Compiler*. Morgan Kaufmann, San Fransisco, CA.

Faraboschi, P., Fisher, J., and Young, C. 2001. Instruction scheduling for instruction level parallel processors. *Proc. IEEE 89,* 11, 1638–1659.

Fog, A. 2012. The micro-architecture of INTEL, AMD and VIA cpus. An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf.

Goodman, J. and Hsu, W. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the International Conference on Supercomputing*.

Govindarajan, R., Yang, H., Amaral, J., Zhang, C., and Gao, G. 2003. Minimum register instruction sequencing to reduce register spills in out-of-order. *IEEE Trans. Comput. 52,* 1, 4–20.

Havanki, W., Banerjia, S., and Conte, T. 1998. Treegion scheduling for wide-issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98).*

Kessler, C. 1998. Scheduling expression DAGs for minimal register need. *J. Comput. Lang. 24*, 1, 33–53.

Langevin, M. and Cerny, E. 1996. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Des. Autom. Electron. Syst. 1*, 4, 443–456.

Malik, A. 2008. Constraint programming techniques for optimal instruction scheduling. Ph.D. thesis, University of Waterloo. https://cs.uwaterloo.ca/~vanbeek/Publications/malik.pdf.

Rim, M. and Jain, R. 1994. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 13,* 4, 451–458.

Shobaki, G. and Wilken, K. 2004. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th International Symposium on Microarchitecture*.

Shobaki, G. 2006. Optimal global instruction scheduling using enumeration. Ph.D. dissertation, Department of Computer Science, UC Davis. http://www.cs.ucdavis.edu/research/tech-reports/2006/CSE-2006-19.pdf.

SHOBAKI, G., WILKEN, K., AND HEFFERNAN, M. 2009. Optimal trace scheduling using enumeration. *ACM Trans. Archit. Code Optim. 5*, 4.

TOUATI, S. 2005. Register saturation in instruction-level parallelism. *Int. J. Parallel Program. 33*, 4, 393–449.

WEICKER, R. AND HENNING, J. 2007. Subroutine profiling results for the CPU2006 benchmarks. *ACM SIGARCH Comput. Archit. News 35,* 1, 102–111.

WINKEL, S. 2007. Optimal versus heuristic global code scheduling. In *Proceedings of the 40th International Symposium on Microarchitecture*.

Received February 2012; revised February 2013; accepted March 2013