Exploring an Alternative Cost Function for Combinatorial Register-Pressure-Aware Instruction Scheduling

GHASSAN SHOBAKI, AUSTIN KERBOW, CHRISTOPHER PULIDO, and
WILLIAM DOBSON, California State University, Sacramento

Multiple combinatorial algorithms have been proposed for doing pre-allocation instruction scheduling with the objective of minimizing register pressure or balancing register pressure and instruction-level parallelism. The cost function that is minimized in most of these algorithms is the peak register pressure (or the peak excess register pressure). In this work, we explore an alternative register-pressure cost function, which is the Sum of Live Interval Lengths (SLIL). Unlike the peak cost function, which captures register pressure only at the highest pressure point in the schedule, the proposed SLIL cost function captures register pressure at all points in the schedule. Minimizing register pressure at all points is desirable in larger scheduling regions with multiple high-pressure points. This article describes a Branch-and-Bound (B&B) algorithm for minimizing the SLIL cost function. The algorithm is based on two SLIL-specific dynamic lower bounds as well as the history utilization technique proposed in our previous work. The proposed algorithm is implemented into the LLVM Compiler and evaluated experimentally relative to our previously proposed B&B algorithm for minimizing the peak excess register pressure. The experimental results show that the proposed algorithm for minimizing the SLIL cost function produces substantially less spilling than the previous algorithm that minimizes the peak cost function. Execution-time results on various processors show that the proposed B&B algorithm significantly improves the performance of many CPU2006 benchmarks by up to 49% relative to LLVM's default scheduler. The geometric-mean improvement for FP2006 on Intel Core i7 is 4.22%.

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:

Ghassan Shobaki, Austin Kerbow, Christopher Pulido, and William Dobson. 2019. Exploring an Alternative Cost Function for Combinatorial Register-Pressure-Aware Instruction Scheduling. *ACM Trans. Archit. Code Optim.* 16, 1, Article 1 (February 2019), 30 pages.
<https://doi.org/10.1145/3301489>

This research was supported by the first author's startup fund from the College of Engineering and Computer Science (ECS) at California State University, Sacramento (CSUS).

Authors' addresses: G. Shobaki, A. Kerbow, C. Pulido, and W. Dobson, California State University, Sacramento, 6000 J Street, Sacramento, CA 95819-6021; emails: {ghassan.shobaki, Kerbow, pulido2, williamdobson}@csus.edu.

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1544-3566/2019/02-ART1

<https://doi.org/10.1145/3301489>

1 INTRODUCTION

Register allocation is a compiler optimization in which the objective is allocating as many *virtual registers* (program variables or compiler generated temporary variables) as possible to CPU registers (physical registers). A virtual register that cannot be allocated to a CPU register is *spilled* to main memory by adding load and store instructions. These loads and stores are called *spill code*. The amount of spill code depends on both the register allocation algorithm and the instruction scheduling algorithm that is run before register allocation (*pre-allocation instruction scheduling*). Pre-allocation instruction scheduling controls the *register pressure* (RP), which is the number of virtual registers that are simultaneously live and thus must be assigned to different physical registers. Therefore, presenting the register allocator with an instruction sequence that has a higher RP is likely to cause the allocator to generate more spill code.

The pre-allocation instruction scheduling problem is further complicated by the presence of another important objective, which is exploiting instruction-level parallelism (ILP) by overlapping the execution of independent instructions. The two objectives of minimizing RP and maximizing ILP inherently conflict with each other. Scheduling more instructions in parallel naturally increases the overlap among virtual registers, thus increasing RP. Therefore, a pre-allocation scheduler must balance these two conflicting objectives.

Both heuristic and combinatorial approaches to register-pressure-aware pre-allocation instruction scheduling have been proposed. Heuristic approaches have been proposed by Goodman and Hsu (1988), Govindarajan et al. (2003), Touati (2005), Barany and Krall (2013), and Makarov (2013). Combinatorial approaches have been proposed by Kessler (1998), Govindarajan et al. (2003), Malik (2008), Barany and Krall (2013), and Domagala et al. (2016).

Excluding the work of Barany and Krall (2013), who used a direct reordering approach inside the register allocator, previous work on combinatorial register-pressure-aware scheduling used cost functions that capture the peak register pressure in the scheduling region. In this article, we introduce an alternative cost function, which is the *Sum of Live Interval Lengths* (SLIL) defined in Section 3. As explained in Section 3, the SLIL cost function captures register pressure across the entire scheduling region, not only at the peak. Thus, minimizing SLIL produces less spilling, on average, than minimizing the peak register pressure. The article describes a combinatorial algorithm for optimizing the proposed SLIL cost function, using a Branch-and-Bound (B&B) approach. The algorithm is implemented in the LLVM open-source compiler and evaluated using SPEC CPU2006 (2006) on two Intel processors (Xeon and Core i7) and using both CPU2006 and MediaBench (Lee et al. 1997) on an ARM Cortex A7 processor. The experimental results show that the proposed B&B algorithm that minimizes SLIL does indeed cause the register allocator to produce significantly less spilling compared to a B&B algorithm that minimizes the peak cost function.

The contributions of this article can be summarized as follows.

- Proposing a new cost function (SLIL) for estimating register pressure during pre-allocation instruction scheduling.
- Describing a B&B algorithm for minimizing the SLIL cost function. The B&B algorithm is based on lower bounds that are developed specifically for the proposed cost function as well as the history utilization technique proposed in our previous work on the Sequential Ordering Problem (SOP) (Shobaki and Jamal 2015). As far as history utilization is concerned, the contribution of the current article is proving that the generalized history utilization technique used in our previous work on the SOP applies to the SLIL cost function and devising sufficient conditions for applying this technique. For the reasons explained in the body of the article, devising a B&B algorithm for minimizing the proposed SLIL cost function is more challenging than devising an algorithm for minimizing the classical peak function.

- Presenting the results of our experimental evaluation of the performance of the proposed B&B algorithm for the SLIL cost function (B&B-SLIL) relative to a B&B algorithm for the peak cost function and LLVM’s default scheduling algorithm. The experimental evaluation includes both spill counts and actual execution times.
- Studying the correlation between the pre-allocation scheduling cost function and the amount of spill code generated by the register allocation algorithm using two different register allocation algorithms.

The rest of the article is organized as follows. Section 2 defines the problem. Section 3 defines the SLIL cost function proposed in this article. Section 4 describes the B&B algorithm for optimizing the SLIL cost function. Section 5 presents the experimental results. Section 6 reviews related work, and Section 7 summarizes the conclusions and outlines future work.

2 PROBLEM DEFINITION

The problem addressed in this article is pre-allocation 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 instructions with this assignment. 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 includes the latency of each instruction and whether it is pipelined or not. The proposed algorithm and its implementation support a general machine model with an arbitrary number of functional units and issue slots per cycle, and arbitrary latencies. It also supports both pipelined and un-pipelined instructions. As detailed in the experimental section, most of the results reported in this article are based on a simple machine model, because using a more accurate model did not actually give better results.

In the pre-allocation scheduling phase, registers in the code are virtual registers. In certain special cases, the code may contain physical registers (for example, if the processor requires certain operands to be stored in specific CPU registers). Each register has a specific data type. 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. Our algorithm and our implementation are general enough to allow an instruction to have an arbitrary number of *Defs* and *Uses*. Given an instruction schedule, the register pressure 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. A register is live at a given point in a schedule if it has been defined but at least one of the instructions that use it has not been scheduled at that point.

We note here that in our implementation, we add an artificial entry node and an artificial exit node to each DDG. This both standardizes the DDG representation and allows us to capture liveness outside the scheduling region. The *Def* set of the entry node includes all the registers that are live at the scheduling region’s entry (live-in registers), and the *Use* set of the exit node includes all the registers that are live at the scheduling region’s exit (live-out registers).

The B&B algorithm described in this article uses the scheduling cost function introduced in our previous work (Shobaki et al. 2013). Given a schedule S , the objective of a combinatorial scheduling

algorithm is to find a schedule that minimizes the following cost function:

$$\text{Cost}(S) = |S| - L_s + w(P - L_p), \quad (1)$$

where $|S|$ is the schedule length, L_s is a lower bound on the schedule length, P is the *register pressure cost*, L_p is a lower bound on the register pressure cost, and w is the *register pressure weight* (RPW). The RPW expresses the weight of register pressure relative to the schedule length. As described in Section 5, the RPW is used to achieve the right balance between ILP and RP.

Various cost functions can be used to represent register pressure. In our previous work, the register-pressure cost function was the *peak excess register pressure* (PERP). The PERP is the maximum value of the *excess register pressure* (ERP) across the schedule. The 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). 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 at any point in the schedule. In the current article, we explore an alternative cost function, which is the SLIL cost function defined in the next section.

3 THE SLIL COST FUNCTION

The PERP was used as a cost function in most previous work, because it was assumed to be the *natural* cost function for a combinatorial formulation of the problem. This assumption is based on the fact that if the PERP can be reduced to zero, the register allocator will not need to generate any spills. This makes PERP the right cost function if the scheduling region has a zero-PERP schedule. However, our experience has shown that there are many scheduling regions for which a zero-PERP schedule does not exist, because the RP in those regions is well above the physical limit. If the RP in a scheduling region is so high that a zero-PERP schedule does not exist, then minimizing the PERP will not be the best approach to minimizing spilling, because PERP does not capture the excess pressure at non-peak points in the schedule. A high-pressure scheduling region may have high RP at multiple points in the region and minimizing the PERP does not necessarily minimize RP at all of these points.

In this article, we explore the Sum of Live Interval Lengths (SLIL) as an alternative cost function that captures RP at all the points in the scheduling region. In a given basic-block schedule, each virtual register has a live interval that consists of one definition and one or more uses. Therefore, each live interval has one *defining instruction* and one or more *using instructions*. The *Live Interval Length* (LIL) is defined as the number of instructions in the instruction sequence that starts with the definition and ends with the last use. Note that the LIL may be greater than the number of defining and using instructions, because in some schedules, these instructions may be interleaved with other instructions that belong to other live intervals (live interval overlap).

For example, if a virtual register has one use and that use is scheduled immediately after the definition, then the LIL will be 2. If two instructions are scheduled between the definition and the only use of that virtual register, then the LIL will be 4 (the definition, the use, and the two instructions in between), and so on. The idea is that a live interval is lengthened by any instruction that does not belong to the live interval but is scheduled between the definition and the last use. The SLIL is the sum of live interval lengths for all virtual registers in a given schedule. A larger SLIL reflects more overlapping among live intervals. When there are multiple register types, the SLIL for each register type is computed separately and the total SLIL is the sum of the SLIL values of all register types. The example in Figure 1 illustrates the calculation of the SLIL cost function for three different schedules.

(a) DDG

Instr.	Regs Defined	Regs Used
C	r_C	None
A	r_A	None
B	r_B	None
D	r_D	r_C
E	None	r_A, r_D
F	None	r_B, r_D

(b) Schedule 1

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

Virtual Reg	Instrs in LI	LIL
r_A	A, B, D, E	4
r_B	B, D, E, F	4
r_C	C, A, B, D	4
r_D	D, E, F	3
SLIL		15

(c) Schedule 2

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

Virtual Reg	Instrs in LI	LIL
r_A	A, B, E	3
r_B	B, E, F	3
r_C	C, D	2
r_D	D, A, B, E, F	5
SLIL		13

(d) Schedule 3

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

Virtual Reg	Instrs in LI	LIL
r_A	A, E	2
r_B	B, F	2
r_C	C, D	2
r_D	D, A, E, B, F	5
SLIL		11

Fig. 1. SLIL calculation.

Figure 1(a) shows the data dependence graph for a small instruction stream consisting of six unit-latency instructions. Instructions A, B, C, and D define virtual registers r_A , r_B , r_C , and r_D , respectively. Each virtual register is used by one instruction, except r_D , which is used by two instructions (E and F). The table to the right of the DDG in Figure 1(a) shows the Def and Use sets of each instruction.

The tables on the left-hand sides of Figures 1(b), 1(c), and 1(d) show three different schedules. The third column in each table shows the registers that are live after the scheduling of each instruction,

and the fourth column shows the register pressure at each point. The peak register pressure (PRP) for Schedule 1 and Schedule 2 is 3, and the PRP for Schedule 3 is 2. If the target machine has two physical registers, then the PERP will be 1 for Schedules 1 and 2 and zero for Schedule 3.

It is important to note here that the PRP and the PERP do not distinguish between Schedules 1 and 2, although Schedule 1 has more interval overlapping and consequently higher RP at more points in the schedule. More specifically, if the target machine has two physical registers, Schedule 1 will have excess RP at two different points, while Schedule 2 will have excess RP at only one point. In a larger scheduling region, a schedule like Schedule 1 presents a harder problem to the register allocator, and this may cause the register allocator to generate more spills.

The tables on the right-hand sides of Figures 1(b), 1(c), and 1(d) show the calculation of the SLIL cost function for each schedule. The second column in each table shows the instructions that constitute the live interval for each virtual register, and the third column shows the length of each live interval (LIL). For example, the live interval for r_A in Schedule 1 (Figure 1(b)) consists of Instructions A, B, D, and E, with a LIL of 4. In this schedule, Instructions B and D appear between the definition of r_A in Instruction A and the last use of r_A in Instruction E. This is an undesirable overlapping of live intervals that increases register pressure and results in more spills. Obviously, the optimal LIL for r_A in this case is 2 (Instruction A followed immediately by Instruction E). Note that Schedule 2 gives a LIL of 3 for r_A and Schedule 3 gives an optimal LIL of 2 for r_A .

The above example shows that minimizing SLIL is more likely to minimize spill code than minimizing PERP, because minimizing SLIL minimizes the overlaps among all intervals in the schedule, while minimizing PERP only minimizes the overlaps that lead to the peak pressure. In a high-pressure scheduling region, the peak-pressure point in the schedule is not the only point at which the register allocator will insert spill code. In the SLIL cost function, every overlap between two or more live intervals adds to the lengths of the overlapping intervals. If there is a schedule that eliminates all overlaps, then the exhaustive search done by the proposed B&B algorithm will find it.

For the reasons explained in the next section, devising an efficient B&B algorithm for minimizing SLIL is much more challenging than devising a B&B algorithm for minimizing PERP. Although the experimental results confirm this relative difficulty, the B&B algorithm that minimizes SLIL produces significantly less spilling than the B&B algorithm that minimizes PERP.

4 ALGORITHM DESCRIPTION

A B&B approach similar to that used in our previous work to minimize the PERP cost function (Shobaki et al. 2013) can be used to minimize the SLIL cost function. However, minimizing the new cost function requires new pruning techniques that are specific to this cost function. In particular, SLIL-specific lower bounds are needed. Furthermore, the history utilization technique used in our previous work must be proven to apply to the SLIL cost function, and sufficient conditions for applying this technique must be devised. In this section, we first give a high-level description of the B&B algorithm. Then we describe the SLIL-specific lower bounds and the sufficient conditions for applying history utilization to the SLIL cost function.

4.1 High-Level Description

The algorithm first constructs an initial heuristic schedule and then computes lower bounds on the schedule length and on the register pressure cost. A lower bound on the schedule length is computed using Langevin and Cerny's algorithm (Langevin and Cerny 1996), while a lower bound on the SLIL cost function is computed using the techniques described in Section 4.2. The cost of the heuristic schedule is then computed using Equation (1). If the cost is zero, then the heuristic schedule is optimal from both ILP and register pressure point of view and the algorithm terminates. If the

cost is non-zero, then the B&B 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 B&B enumerator explores one schedule length at a time starting at the schedule-length lower bound. The search traverses a decision tree (or an *enumeration tree*) in a depth-first manner. In this tree, the root represents an empty schedule, each internal node represents a *partial schedule*, and each leaf node represents a *complete schedule*. A complete schedule below a given internal tree node consists of a *prefix schedule* (the instructions that have been scheduled at that node) and a *suffix schedule* (the instructions that are yet to be scheduled in the successor tree nodes below that node). The cost of the prefix schedule is called the *prefix cost* and the cost of the suffix schedule is called the *suffix cost*. Since the search is done in a depth-first manner, only the nodes on the path from the root to the current node need to be kept in memory.

To develop an efficient B&B algorithm for a given minimization problem, tight lower bounds are needed. Lower bounds are needed before and during enumeration. Before enumeration, tight *static lower bounds* (SLBs) are needed to filter out the instances whose initial heuristic solutions are already optimal. During enumeration, tight *dynamic lower bounds* (DLBs) are needed to effectively prune the sub-trees that cannot have better solutions than the current best solution. These lower bounds are described in the next two sections.

4.2 Static Lower Bounds

In the proposed algorithm, the following SLIL-specific SLBs are used.

4.2.1 Basic Lower Bound. A basic lower bound on the SLIL of a given DDG can be computed by simply counting the instructions that define and use each virtual register. For example, in the DDG of Figure 1, each of the live intervals of registers A, B, and C has one defining instruction and one using instruction. This gives a lower bound of two on each of these three LILs. The live interval of D has one defining instruction and two using instructions, which gives a lower bound of 3 on the LIL of D. Summing the four LIL lower bounds gives a SLB of 9 on the SLIL of that DDG.

4.2.2 Common-Use Lower Bound. The live intervals of two or more virtual registers must overlap if the virtual registers have a *common use*. A common use of two virtual registers is an instruction that has both registers in its Use set. For example, in the DDG of Figure 1, Instruction E is a common use of registers r_A and r_D . Therefore, the live intervals of r_A and r_D must overlap. The unavoidable overlap between the live intervals of r_A and r_D implies that one of these two live intervals must have an extra instruction that is not one of its using instructions. This allows us to add one to the basic lower bound, thus computing a tighter lower bound of 10 on the SLIL for the DDG of Figure 1. Similarly, Instruction F is a common use of virtual registers r_B and r_D , which implies that the live intervals of r_B and r_D must overlap, thus allowing us to add another one to the lower bound and compute a common-use-based SLB of 11 on the SLIL of the DDG in Figure 1. Note that this tighter lower bound leads to proving the optimality of Schedule 3 in Figure 1(d).

The common-use rule for computing tighter lower bounds may be generalized to account for an arbitrary number of virtual registers. If x virtual registers have a common use, then the virtual register that appears first in the schedule will have $x-1$ extra instructions in its live interval, the virtual register that appears second will have $x-2$ extra instructions in its live interval, and so on. Therefore, the total number of extra instructions that can be added to the basic SLB is

$$\sum_{i=1}^{x-1} i = \frac{x(x-1)}{2}. \quad (2)$$

The above formula assumes that the defining instruction of each virtual register in the common-use set is not a using instruction of any other virtual register in the set. If virtual registers r_1 and

r_2 have a common use, but the defining instruction of virtual register r_2 uses virtual register r_1 , then the overlap is unavoidable. However, that does not imply any extra instructions, because r_1 must be scheduled before r_2 , and r_2 already belongs to the live interval of r_1 .

4.2.3 Dependence-Based Lower Bound. An even tighter lower bound on the SLIL may be computed by accounting for non-data dependencies in the DDG. Ideally, a DDG in the pre-allocation pass has only data dependencies. Anti and output dependencies should not be introduced until register allocation is done. In practice, however, the DDG in the pre-allocation pass may have some anti and output dependencies, because some virtual registers may have already been assigned to specific physical registers before pre-allocation scheduling. One reason for such an assignment is satisfying the target processor's requirement that a certain operand must be stored in a specific physical register. In addition to anti and output dependencies, the compiler may introduce other non-data dependencies for various reasons. Non-data dependencies in the DDG may be used to compute tighter lower bounds as follows.

Given a defining instruction x and a using instruction y , every instruction that is a *recursive predecessor* of y and a *recursive successor* of x in the DDG must be scheduled between x and y . Instruction u is a recursive predecessor of instruction y in a given DDG if there is a path from u to y in the DDG. Similarly, instruction v is a recursive successor of x if there is a path from x to v in the DDG. The recursive predecessor and successor lists of each instruction in the DDG are constructed by computing the transitive closure of the given DDG (Cormen et al. 2009). A tighter lower bound on the length of a given live interval L with defining instruction x may be computed by checking the instructions that belong to the intersection of the recursive successor list of x and the recursive predecessor list of every using instruction y , and then adding those instructions to L if they do not already belong to L .

4.3 Dynamic Lower Bounds

The techniques described in the previous section for computing static lower bounds before enumeration can also be used to compute dynamic lower bounds during enumeration and prune the enumeration tree. If the DLB at the current node is not less than the best cost found so far, then the sub-tree below the current node may be pruned. At any given point during enumeration, a live interval can be in one of three possible states: *open* (defining instruction has been scheduled but not all using instructions have been scheduled yet), *closed* (defining instruction and all using instructions have been scheduled) or *not started* (defining instruction has not been scheduled yet). Whenever the B&B enumerator adds an instruction to the current partial schedule, it checks all open live intervals and increments the length of each live interval that the scheduled instruction does not belong to or does not *possibly* belong to. Instruction x possibly belongs to an open live interval if the defining instruction of the open live interval has a common use with instruction x . In other words, the length of an open live interval is incremented whenever the newly scheduled instruction is an instruction that was not counted in the SLB calculation of this interval.

DLB calculations are illustrated in Figure 2. Figure 2 shows two partial schedules that a B&B scheduler may construct for the DDG in Figure 1. Figure 2(a) shows the enumeration of Partial schedule $\langle C, A, B \rangle$ and Figure 2(b) shows the enumeration of Partial schedule $\langle C, D, A \rangle$. In both cases, it is assumed that the initial feasible schedule that was constructed before enumeration was Schedule 2 shown in Figure 1(c) with a SLIL of 13. Thus, the B&B enumerator will search for a schedule with a cost less than 13. Any schedule with a cost of 13 or higher is not interesting. Therefore, whenever the B&B enumerator constructs a partial schedule with a DLB of 13 or higher, the subtree below the current node is pruned and the B&B enumerator backtracks.

First, consider the partial schedule $\langle C, A, B \rangle$ in Figure 2(a). At the root node, no instruction has been scheduled and the DLB is equal to the SLB of 11. When the B&B enumerator steps forward by

Fig. 2. Dynamic lower bounds.

scheduling Instruction C, that does not change the DLB, because it does not add any extra instruction to any open live interval. The only open interval at that point is the interval of virtual register r_C , and Instruction C is an original member of that interval. In the second step, the enumerator schedules Instruction A and that results in incrementing the length of the C live interval by 1, because Instruction A does not belong or possibly belong to that open interval. Instruction A does not possibly belong to the C live interval, because A and C do not have a common use. This leads to a DLB of 12, which means that every schedule that may be found below Node 2 will have a cost of 12 or higher. Since the best cost at that point is 13 (the cost of the initial feasible schedule), no pruning takes place at Node 2, and the enumerator steps forward by scheduling Instruction B.

When Instruction B is scheduled, the LILs of both C and A are incremented, because Instruction B does not belong or possibly belong to either live interval. This leads to incrementing the DLB by 2, thus producing a DLB of 14. At this point, the DLB is greater than the best-known cost of 13, which leads to pruning the sub-tree below Node 3 and backtracking to Node 2.

In Figure 2(b), the B&B scheduler constructs the partial schedule $\langle C, D, A \rangle$. Initially, the DLB is equal to the SLB of 11 at the root node, because no instructions are scheduled. Similar to the partial schedule of Figure 2(a), the DLB at Node 1 is equal to the SLB, because Instruction C already belongs to its own live interval. When Instruction D is scheduled next, this does not change the DLB, because D belongs to the live interval of C. Instruction D closes the live interval of C, and the only open interval at Node 2 is the interval of D. When Instruction A is scheduled next at Node 3, the length of A's live interval is not incremented, because A belongs to its own live interval. The length of D's live interval is not incremented either, because A and D have a common use, and A possibly belongs to D's live interval (it was counted in the SLB). Thus, the DLB at Node 3 remains 11, and no pruning takes place. If the B&B scheduler continues to explore the subtree below Node 3 in Figure 2(b), it will eventually discover the schedule $\langle C, D, A, E, B, F \rangle$ (Schedule 3 in Figure 1(d)), which has an optimal cost of 11.

It is noted that the DLBs described in this section were not needed in our previous work on minimizing the PERP (Shobaki et al. 2013). Our B&B algorithm for minimizing the PERP used the PERP at the current node as a DLB, and that greatly simplified the algorithm. The current algorithm for minimizing SLIL cannot use the SLIL at the current node as a DLB, because that would be too loose and won't lead to effective pruning. The PERP at the current node can be a tight DLB, because the schedule may reach its peak pressure early, while the SLIL at the current node is unlikely to

be a tight DLB, because it does not capture the cost of unscheduled instructions, hence the need for the DLBs described in this section.

4.4 History Utilization

History domination is a pruning technique that was first introduced in a combinatorial algorithm for solving the superblock scheduling problem (Shobaki and Wilken 2004). The technique was then used to solve the pre-allocation scheduling problem with the PERP cost function (Shobaki et al. 2013). The main idea behind this technique is storing information about previously visited tree nodes in a *history table* and then using that information to speed up the processing of the *similar* tree nodes that are encountered later. Two tree nodes x and y are similar if the set of scheduled instructions in x 's prefix schedule is equal to the set of scheduled instructions in y 's prefix schedule. If two nodes x and y are similar, then the remaining sub-problem to be solved below x is the same as the remaining sub-problem to be solved below y .

A more general form of the technique, called *history utilization*, was used in a B&B algorithm for the SOP (Shobaki and Jamal 2015). Unlike the original history domination technique, which is solely used to prune the sub-tree below the current tree node if its prefix cost is not better than the prefix cost of the history node, the generalized history utilization technique may construct a better solution than the current best solution without having to explore the sub-tree below the current node. If the conditions described below are satisfied, then the best suffix schedule below the current node is the same as the best suffix schedule below the history node. Therefore, if the prefix schedule at the current node has a lower cost than the prefix schedule in the history table, concatenating the suffix schedule below the history node (stored in the table) to the current prefix schedule will give a better schedule without having to explore the sub-tree below the current node.

This extended form of history utilization is used in our current work to minimize the SLIL cost function but with additional conditions that account for the relative complexity of the pre-allocation scheduling problem. The pre-allocation scheduling problem is more complex than the SOP, because the objective function in pre-allocation scheduling is a weighted sum of schedule length and register pressure, while the cost function in the SOP is a single number. Therefore, similar to the B&B algorithm for minimizing PERP, the algorithm for minimizing SLIL first checks latency and resource constraints before checking the register pressure cost.

As detailed in previous work (Shobaki et al. 2013; Shobaki 2006), latency and resource constraints are checked first at each node to determine if the remaining sub-problem below the current node is less constrained than the remaining sub-problem below the history node. If the sub-problem below the current node is less constrained, then a better suffix schedule may be found below the current node. Therefore, the sub-tree below the current node must be explored and no pruning can be done. If the current node is *not* less constrained than the history node from both resource and latency points of view, then the current node is *dominated* by the history node, in which case the register pressure cost will be checked to determine if history utilization can be applied.

In our previous work, we presented an argument showing that history domination can be applied to the problem of minimizing the PERP (Shobaki et al. 2013). In this article, we modify the argument to show that history domination can be applied to the problem of minimizing SLIL. Minimizing SLIL is different than minimizing PERP for the following two reasons.

- (1) In minimizing PERP, the cost of a complete schedule below a given node is the *maximum* of the prefix cost and the suffix cost, while in minimizing SLIL, the cost of a complete schedule below a given node is the *sum* of the prefix cost and the suffix cost.
- (2) In minimizing PERP, the suffix cost is completely independent of the instruction order in the prefix schedule, while in minimizing SLIL, the lengths of open live intervals are dependent on the instruction order in the prefix schedule.

Fig. 3. History utilization example.

The first difference between PERP and SLIL does not affect correctness, because history domination will be correct (the current node can be safely pruned) as long as the current node cannot have below it a better suffix schedule than the best suffix schedule below the history node. Pruning can be done whether the suffix cost is added to the prefix cost (when the objective is minimizing SLIL) or compared with the prefix cost to find the maximum of the two costs (when the objective is minimizing PERP). Therefore, it remains to show that the second difference (dependence of live interval lengths on the prefix schedule) does not affect correctness either.

It turns out that although the length of an open live interval is dependent on the prefix schedule, the cost of the suffix schedule will always be independent of the prefix schedule. This is due to the similarity between the current node and the history node, which implies that open live intervals are the same for both nodes. Since every instruction in the suffix schedule adds one to the length of every open live interval, the added value to the suffix cost below the current node will be equal to the added value to the suffix cost below the history node.

To illustrate this, consider the two prefix schedules in Figure 3 that may be constructed by a B&B scheduler for the DDG in Figure 1. Node 3 in Figure 3(a) is a history node with partial schedule $\langle C, A, B \rangle$, and Node 7 in Figure 3(b) is a current node with partial schedule $\langle C, B, A \rangle$. Clearly, the two nodes are similar, as scheduled instructions in both prefix schedules are the same.

The figure shows the lengths of open live intervals, the prefix cost and the DLB at each node in the figure. It is important to note here that the prefix cost is the SLIL of the prefix schedule, while the DLB is a lower bound on the SLIL of any *complete* schedule below the node. At the history node (Node 3), the open live intervals are the intervals of virtual registers r_A , r_B , and r_C . The LIL of each open live interval is shown to the right of Node 3, and the sum of these LILs gives a prefix cost of 6. Figure 3(b) shows similar information for the current node (Node 7). It is noted that each prefix schedule gives different LILs for A and B. However, because the two nodes are similar, the set of open live intervals is the same. Therefore, an instruction in the suffix schedule will add exactly the same value to every open live interval. For example, if Instruction D is scheduled next in the suffix schedule (which is the only feasible option in this case), it will close live interval C and add one to the LILs of A, B, and C.

This shows that if the remaining sub-problem below the current node is not less constrained than the remaining sub-problem below a similar history node, the current node cannot have below

it a better suffix schedule than the best suffix schedule below the history node. Depending on the relation between the current prefix cost and the history prefix cost, one of the following three actions will be taken.

- (1) If the prefix cost at the current node is not smaller than the prefix cost of the history node, then the sub-tree below the current node may be pruned, because the total cost of any schedule below the current node cannot be better than the total cost of the best schedule below the history node.
- (2) If the prefix cost at the current node is smaller than the prefix cost at the history node but the *possible improvement* is not greater than the *needed improvement*, then the current node may also be pruned. The possible improvement is the difference between the prefix cost at the history node and the prefix cost at the current node, and the needed improvement is the difference between the best total cost below the history node (or a DLB on it) and the best cost found so far. If the possible improvement is not greater than the needed improvement, then the current node cannot have below it a better schedule than the best schedule found so far.
- (3) If the prefix cost at the current node is smaller than the prefix cost at the history node and the possible improvement at the current node is greater than the needed improvement, then a new best schedule can be constructed directly by concatenating the suffix schedule below the history node (if known and stored in the history table) to the current node's prefix schedule. This action is called *suffix concatenation*. Note that the best suffix below the history node may not be known, because the search below the history node may have always backtracked without reaching a leaf.

5 EXPERIMENTAL RESULTS

The B&B algorithm described in this article was implemented and integrated into the LLVM Compiler (Version 3.9) as an alternative pre-allocation scheduler. LLVM's Clang and Clang++ front ends were used for the C and C++ benchmarks, respectively, and Dragon Egg was used for the FORTRAN benchmarks. In all tests, except the special test reported in Section 5.6, the front end was invoked with the `-O3` option. At this optimization level, LLVM invokes its greedy global register allocator.

Three different target processors were used in our experimental evaluation:

1. Intel Xeon E5-1620 v4 dual-core processor running at 3.5GHz
2. Intel Core i7-7700K processor running at 4.2GHz
3. ARM Cortex A7 processor running at 960MHz

Xeon and Core i7 are out-of-order processors, while ARM A7 is an in-order processor. The SPEC CPU2006 benchmarks (2006) were used on all processors, and the MediaBench benchmarks (Lee et al. 1997) were used only on ARM. The benchmarks were cross-compiled for ARM on the Xeon processor. Due to cross-compilation issues, some MediaBench and CPU2006 benchmarks could not be compiled for ARM. In particular, none of the FORTRAN benchmarks in FP2006 could be cross-compiled for ARM due to compatibility issues between our version of Dragon Egg and the ARM back end in our version of LLVM. The tests were run on Ubuntu 16.04. All tests were compiled and executed using a single thread.

The newer versions of LLVM, including the version that we used, have a machine-level pre-allocation scheduler that does scheduling at a lower level representation of the code than that used in the earlier versions. Our B&B scheduler was integrated as an alternative scheduler to LLVM's machine-level pre-allocation scheduler, which is a heuristic scheduler. The comparative results

Table 1. Benchmark Statistics

	FP2006	INT2006	MEDIABENCH
Number of benchmarks	17	12	12
Number of functions	23,147	24,827	569
Number of scheduling regions	563,003	358,100	10,483
Avg. scheduling regions per function	24.3	14.4	18.4
Avg. Instructions per scheduling region	8.0	5.6	4.9
Max. Instructions per scheduling region	6,263	666	1,123

Table 2. B&B Scheduling Statistics for CPU2006 on Xeon.
Time Limit = 15ms/instr, Unit Latencies

	STAT	FP2006	INT2006
1	Total regions processed	563,003	358,100
2	Regions passed to B&B	303,305 (53.9%)	152,463 (42.6%)
3	Regions optimal and improved	190,355 (62.8%)	88,850 (58.3%)
4	Regions optimal and not improved	101,532 (33.5%)	63,187 (41.4%)
5	Regions timed out and improved	9,909 (3.3%)	376 (0.3%)
6	Regions timed out and not improved	1,509 (0.5%)	50 (0.0%)
7	Largest optimal region	256	274
8	Largest improved region	3,675	666
9	SLIL cost improvement	38.1%	62.6%

in this section use LLVM's machine scheduler as a baseline. For a fair comparison between the proposed pre-allocation scheduler and LLVM's pre-allocation scheduler on ARM, LLVM's post-allocation scheduler was turned off.

It should be noted that the scheduling regions in LLVM's machine scheduler are not necessarily basic blocks. In some cases, the machine scheduler may divide a basic block into multiple scheduling regions to facilitate the construction of the DDG.

5.1 Benchmark Statistics

The benchmark suites used in our experiments are SPEC CPU2006, including INT2006 and FP2006, and MediaBench. Table 1 shows some statistics about these benchmarks:

5.2 B&B Scheduling Statistics

The proposed B&B algorithm that minimizes the SLIL cost function (B&B-SLIL) was applied to CPU2006 on all three processors and to MediaBench only on ARM. In this section, we present statistics at the scheduling-region level. Due to space limitations, these statistics are presented only for the Xeon processor. The statistics for Core i7 and ARM were similar. For Xeon, the time limit for each scheduling region was set to 15 ms per instruction. So, a scheduling region with 100 instructions was given a limit of 1.5 seconds. Since this particular test focuses on minimizing RP, all instruction latencies were set to unity (to effectively ignore ILP) and the RPW in Equation (1) was set to 1. LLVM's heuristic machine scheduler was used to generate the initial feasible schedule. The cost of the heuristic schedule was computed using Equation (1). If the cost is zero, the heuristic schedule is optimal. Therefore, the B&B scheduler was invoked to search for a lower cost schedule only if the cost computed by Equation (1) was greater than zero. Table 2 shows the scheduling statistics when the proposed B&B algorithm was applied to all functions in CPU2006.

It is noted that in our implementation, suffix concatenation significantly reduced the number of enumerated tree nodes but did not give a significant reduction in solution time, because it increased the processing time per node. So, suffix concatenation was disabled in all experiments.

First, we comment on the results for FP2006. Row 2 shows that 54% of the scheduling regions were passed to the B&B scheduler, because they did not have provably optimal heuristic schedules. The remaining 46% of the regions had heuristic schedules with zero SLIL costs.

Rows 3 and 4 show that 96.3% of the scheduling regions passed to the B&B scheduler were scheduled optimally, 62.8% were improved relative to the heuristic and 33.5% were not improved. Row 5 shows that 3.3% of the regions were not scheduled optimally but were improved relative to the heuristic.

Combining the number of regions in Rows 5 and 6 gives the total number of regions that timed out, which is 11,418 (3.8%). A closer look at these 11,418 regions reveals that these are, on average, larger scheduling regions containing 19.2% of the total number of instructions. This is expected, because larger scheduling regions are harder to schedule optimally. The spilling statistics in the next section show that in spite of this significant number of timeouts, the proposed B&B-SLIL scheduler produces significantly less spilling than the LLVM scheduler and the B&B scheduler for the PERP cost function.

Row 7 shows that the proposed algorithm was powerful enough to optimally schedule a region with 256 instructions and Row 8 shows that it improved the schedule of a region with 3,675 instructions. Row 9 shows that the B&B scheduler improved SLIL by 38%, which is quite significant. However, it is important to note that SLIL improvements do not necessarily result in spill code reductions, because minimizing SLIL may eliminate many interval overlaps that are below the physical-register limit and thus do not affect the amount of spilling.

Comparing the INT results with the FP results shows that the main difference is that the INT scheduling regions are, on average, smaller. This explains why the number of regions that timeout (Rows 5 and 6) is significantly less for INT.

5.3 Spill Code Statistics

In this section, we evaluate the performance of the proposed B&B-SLIL algorithm relative to other algorithms in terms of the amount of spilling generated by the register allocator. The evaluated algorithms are as follows:

- LLVM’s heuristic machine scheduler (LLVM)
- Our B&B algorithm for the proposed SLIL cost function and ILP ignored by setting all latencies to unity (BB_SLIL_RP).
- Our B&B algorithm for the proposed SLIL cost function and ILP accounted for using the latency values set by LLVM (BB_SLIL_RP&ILP).
- Our B&B algorithm for the PERP cost function and ILP ignored by setting all latencies to unity (BB_PERP_RP).
- Our B&B algorithm for the PERP cost function and ILP accounted for using the latency values set by LLVM (BB_PERP_RP&ILP).
- A Critical-Path (CP) list scheduling algorithm that schedules for ILP and ignores register pressure (CP_ILP) (Shobaki et al. 2015).

The purpose of including the CP scheduler is to measure the amount of spilling that would be generated if RP was ignored during per-allocation scheduling. This can be viewed as the worst-case behavior from RP point of view.

For the B&B algorithms, the initial feasible schedule was generated using LLVM’s machine scheduler. Each algorithm under study was applied to all the functions in each benchmark, and

Table 3. Spill-code statistics on Intel Xeon. Time limit = 15 ms/instr

	STAT	FP2006	INT2006
1	Total functions	23,147	24,827
2	Total instructions	1,996,130	2,839,876
3	Functions with spills	8,474 (36.6%)	3,032 (12.2%)
4	Sum of minima (SOM)	281,946	20,984
5	Avg. spills per function	12.2	0.85
6	Avg. spills per spilling function	33.3	6.9

the number of live ranges spilled by LLVM's default register allocator was used as a metric for evaluating each scheduling algorithm's effectiveness at reducing RP. At -O3, LLVM invokes its greedy global register allocator.

To evaluate the performance of the register-pressure-aware algorithms, we use the metric introduced in our previous work, which is an experimental lower bound on the gap between the performance of the algorithm under study and optimal performance (Shobaki et al. 2015). An experimental upper bound on the optimal sum of spills for a large set of functions is computed by applying multiple algorithms to each function, taking the minimum number of spills per function across all algorithms (best result) and then summing these minima over all functions. For a given set of functions F and a given set of algorithms A , the Sum of Minima (SOM) is defined as

$$SOM(F, A) = \sum_F \min_A(spills(f, a)), \quad (3)$$

where $spills(f, a)$ is the number of spills generated by the register allocator when algorithm $a \in A$ is used to schedule the scheduling regions in function $f \in F$. As explained in previous work, the SOM is an upper bound on the optimal sum of spills for the given function set F .

To evaluate the performance of a given algorithm on a given set of functions F , we compute the difference between the sum of spills produced by that algorithm on F and $SOM(F, A)$ taken over a set of algorithms A . For a given algorithm a in a set of algorithms A , the difference between a 's spill sum and the SOM is

$$ExtraSpills(a, A) = \sum_F spills(f, a) - SOM(F, A). \quad (4)$$

$ExtraSpills(a, A)$ is a lower bound on the size of the gap between the number of spills produced by algorithm a and the optimal number of spills. In the results below, $ExtraSpills$ is also referred to as the *optimality gap*.

Table 3 shows the spill-code statistics on the Xeon processor. To compute the tightest possible experimental upper bound on the optimal SOM, a large set of algorithms was used (45 algorithms), including our B&B algorithm with different parameter settings and a collection of heuristics from our previous work (Shobaki et al. 2015).

Rows 1 and 2 in Table 3 show the total number of functions and instructions, respectively, in each of FP2006 and INT2006. Row 3 shows the number of functions that have spills. A function may have spills with some scheduling algorithms but not with other algorithms. The entry in Row 3 is the number of functions that have spills with *at least* one scheduling algorithm. The SOM in Row 4 was computed using Equation (3) for the complete set of algorithms. Row 5 shows the average spills per function (Row 4 divided by Row 1), and Row 6 shows the average spills per spilling function (Row 4 divided by Row 3).

Comparing Rows 2 and 4 shows that the number of spills is an order of magnitude less than the total number of instructions on FP2006 and two orders less than the total instruction count on

(a) Algorithm ranking and optimality gaps for FP2006

ALGORITHM	EXTRA SPILLS	%FUNCS AT MIN	MAX EXTRA PER FUNC
BB_SLIL_RP	9,820 (3.48%)	65.7%	150
BB_PERP_RP	12,128 (4.30%)	62.3%	174
LLVM	13,137 (4.66%)	61.4%	182
BB_SLIL_RP&ILP	16,412 (5.82%)	62.4%	325
BB_PERP_RP&ILP	18,327 (6.50%)	58.2%	325
CP_ILP	51,627 (18.31%)	54.6%	2244

(b) Algorithm ranking and optimality gaps for INT2006

ALGORITHM	EXTRA SPILLS	%FUNCS AT MIN	MAX EXTRA PER FUNC
BB_SLIL_RP	1,051 (5.01%)	82.9%	22
BB_SLIL_RP&ILP	1,187 (5.66%)	82.3%	37
BB_PERP_RP	1,294 (6.17%)	80.1%	44
LLVM	1,315 (6.27%)	80.0%	44
BB_PERP_RP&ILP	1,469 (7.00%)	79.0%	37
CP_ILP	2,063 (9.83%)	78.3%	74

Fig. 4. Algorithm ranking and optimality gaps on Intel Xeon.

INT2006. The execution-time results show that although spills constitute only a small percentage of the code size, reducing spilling in hot functions can have a significant impact on performance.

Figure 4 shows the optimality gaps of the six algorithms under study for FP2006 and INT2006 on Intel Xeon. The optimality gap (EXTRA SPILLS) is also shown in the second column of the table next to each graph. EXTRA SPILLS is the number of spills produced by each algorithm relative to the SOM (Row 4 in Table 3). The algorithms in the table and in the graph are sorted by EXTRA SPILLS. Column 3 in each table shows the percentage of functions for which each algorithm produced the minimum amount of spilling, and Column 4 shows the maximum amount of extra spills that each algorithm produced in a single function (the algorithm's worst-case behavior).

First, we comment on the FP results in Figure 4(a). Overall, the numbers in the table show that by all three metrics, the proposed B&B-SLIL algorithm in RP-only mode (BB_SLIL_RP) is statistically superior to all other algorithms under study. The second-best algorithm is the B&B algorithm for minimizing PERP. LLVM's algorithm ranks third with 13,137 extra spills (4.66% more than the SOM). The next two rows show that when the B&B algorithm balances RP and ILP, it produces more spills than the LLVM scheduler. This indicates that the LLVM scheduling algorithm is biased

toward minimizing RP on the Intel Xeon target, which is a reasonable approach on a powerful out-of-order target. However, the execution-time results in the next section show that the proposed B&B-SLIL algorithm in balanced mode (BB_SLIL_RP&ILP) outperforms LLVM's scheduler on many benchmarks, because LLVM's scheduler misses some ILP exploitation opportunities.

The last bar in Figure 4(a) shows that the CP heuristic, which is a pure-ILP heuristic that does not consider RP, produces 51,627 extra spills (18% more than the SOM). This is a quantitative confirmation of the intuitively expected result that ignoring RP can result in excessive spilling. The fourth column in the table shows that, in the worst case, the CP heuristic produces 2,244 extra spills in one function. The execution-time results in the next section show that such excessive spilling may occur in hot functions and result in a substantial performance degradation.

It is noted here that although the proposed B&B-SLIL algorithm produces the minimum amount of spilling compared to other algorithms in this experiment, it produces 9,820 extra spills (3.48%) relative to the SOM, which is still a significant gap. Column 3 shows that the proposed B&B-SLIL algorithm produces the minimum amount of spilling in only 65.7% of the functions, and Column 4 shows that, in the worst-case, it produces 150 extra spills in a single function. These metrics clearly show that the gap between the best B&B algorithm and optimality is still significant. This gap is attributed to the following factors:

- (1) The proposed B&B-SLIL scheduler timed out on a significant number of scheduling regions. As shown in Table 2, 11,418 regions timed out, and these regions have 19.2% of the total instructions in FP2006. It is not clear how much spill code reduction would be achieved if all of these regions were scheduled optimally. Answering this question requires making substantial enhancements to the proposed B&B-SLIL algorithm, such as developing a parallel version of the algorithm, which we plan on pursuing in future work. However, the fact that the proposed B&B-SLIL algorithm is statistically superior to all other algorithms in spite of the significant number of timeouts shows the great potential for minimizing the proposed SLIL cost function.
- (2) Although the SLIL cost function explored in this article is statistically superior to the PERP cost function in terms of the amount of spill code, the correlation between SLIL and spill code is still not perfect. Achieving perfect correlation between the scheduling cost function and the amount of spilling produced by the register allocator is unlikely, because of the complexity of the register allocation problem. As discussed in Section 5.6, the correlation is not perfect even between different register allocation algorithms. For any given piece of code, different register allocation algorithms give different amounts of spilling. This leads to the conclusion that achieving the best correlation between a specific register allocation algorithm and the instruction order requires integrating scheduling into the register allocation algorithm, which is an interesting problem for future work.

Comparing the INT2006 results in Figure 4(b) with the FP2006 results in Figure 4(a) shows that the INT benchmarks generally have easier scheduling regions with fewer spills. Spill counts are an order of magnitude lower in the INT benchmarks, the algorithm success rates (%FUNCS AT MIN) are significantly higher, and the maximum amount of extra spilling per function is significantly lower. Interestingly, the ranking of the algorithms is the same as that for the FP benchmarks, except that the proposed algorithm in balanced mode (BB_SLIL_RP&ILP) ranks second in Figure 4(b). It outperforms the B&B algorithm for PERP in RP-only mode. This is attributed to the fact that the INT benchmarks have lower degrees of ILP, thus making it easier for the balanced scheduler to exploit ILP without compromising RP.

Figures 5 and 6 show the optimality gaps for FP2006 on Core i7 and ARM A7, respectively. The ranking of the algorithms on i7 is the same as that on Xeon, while the ranking on ARM is slightly

Fig. 5. Algorithm ranking and optimality gaps for FP2006 on Intel Core i7. SOM = 285,708.

Fig. 6. Algorithm ranking and optimality gaps for FP2006 on ARM Cortex A7. Total functions: 9,806, SOM = 19,226.

Table 4. Comparison between SLIL and PERP.
Processor: Intel Xeon. Time Limit: 15ms/instr

		SLIL	PERP
1	Regions that timeout	11,418	5,968
2	Instructions in timeout regions	597,621	334,379
3	Extra spills relative to SOM	9,820	12,128

different with the balanced form of B&B-SLIL outperforming LLVM. A time limit of 10ms/instr was used on i7, because the i7 processor is a bit faster than the Xeon processor. A time limit of 15ms/instr was used in cross-compiling for ARM on the Xeon processor.

Since the objective of this test is comparing the ranking of the algorithms on Core i7 and ARM A7 with that on Xeon, only smaller algorithm sets were used to compute the SOMs (10 algorithms for Core i7 and 15 for ARM), thus producing larger SOMs and smaller optimality gaps. Clearly, this does not affect the ranking.

Table 4 shows a direct comparison between the proposed B&B algorithm for minimizing SLIL and the previous B&B algorithm for minimizing PERP. Row 1 shows that the number of regions that timeout when the cost function is SLIL is substantially greater than the number of regions that timeout when the cost function is PERP. The same applies to the number of instructions in these regions. Yet, the B&B algorithm for SLIL produces 2,308 fewer spills. This result shows that the SLIL cost function has a significantly greater potential than the PERP cost function although it is a harder cost function to minimize. Developing more advanced algorithmic techniques for minimizing SLIL is expected to lead to an even larger spill-code gap between SLIL and PERP.

Table 5 shows the spill-code statistics when the algorithms under study are applied only to the hot functions in FP2006 on the Xeon processor. Figure 7 shows the optimality gaps and algorithm ranking. Hot function selection is the same as that used in our previous work (Shobaki et al. 2013),

Table 5. Spill Code Statistics in the Hot Functions in FP2006. Processor: Intel Xeon. Time Limit: 15ms/inst

	STAT	VALUE
1	Total hot functions	71
2	Functions with spills	61 (86%)
3	Sum of minima (SOM)	10,606
4	Avg. spills per hot function	149.4
5	Avg. spills per spilling hot function	173.9

ALGORITHM	EXTRA SPILLS	%FUNCS AT MIN	MAX EXTRA PER FUNC
BB_SLIL_RP	197 (1.9%)	45.9%	24
BB_PERP_RP	448 (4.2%)	34.4%	98
LLVM	504 (4.8%)	36.1%	98
BB_SLIL_RP&ILP	616 (5.8%)	32.8%	67
BB_PERP_RP&ILP	749 (7.1%)	23.0%	94
CP_ILP	2668 (25.2%)	27.9%	739

Fig. 7. Algorithm ranking and optimality gaps for hot functions in FP2006 on Intel Xeon.

which was based on the profiling information published by Weicker and Henning (2007). The SOM in this test was computed using 36 algorithms (less than what was used in the all-function experiment in Table 3 and Figure 4). The statistics in Table 5 show that most hot functions have spills and that the average number of spills per functions is significantly higher than that in the all-function experiment (149 in Table 5 compared to 12 in Table 3). This is consistent with the fact that hot functions tend to be larger functions containing more complex computation.

The ranking of the algorithms in Figure 7 is the same as that seen in the all-function experiment (Figure 4). The proposed B&B-SLIL algorithm in RP-only mode produces the least extra spills in hot functions. It is noted that in this test the difference between SLIL and PERP is much more significant than it was in the all-function test (1.9% extra spills for SLIL, compared to 4.2% extra spills for PERP). This is attributed to the fact that a hot function tends to have larger scheduling regions with higher RP values. As explained in Section 3, the motivation behind using SLIL is capturing multiple high-pressure points in larger scheduling regions.

It is also noted that the performance of the CP heuristic on hot functions is significantly poorer than it was in the all-function experiment. In hot functions, CP produced 25% extra spills, while in the all-function experiment, it produced 18% extra spills relative to the SOM. This is consistent with the fact that the larger scheduling regions in hot functions tend to have higher degrees of ILP. Being unaware of RP, the CP scheduler exploits that ILP at the expense of increasing RP. As shown in the next section, such increase in RP can degrade a benchmark's execution-time performance by up to 36%.

Table 6. Settings for Execution-Time Tests

PROCESSOR	BENCHMARKS	MACHINE MODEL	HEURISTIC	RPW	TIME LIMIT
Xeon	CPU2006	Simple	LLVM	1.0	15ms/instr
Core i7	CPU2006	Simple	LUC	1.0	10ms/instr
ARM A7	CPU2006	Simple	LLVM	0.008	15ms/instr
ARM A7	MediaBench	Accurate	LLVM	0.012	10ms/instr

5.4 Execution Times

The spill-code statistics reported in the previous section show that the proposed B&B-SLIL algorithm produces significantly less spilling than other algorithms. In this section, we study the impact of spill-code reduction on the benchmarks' actual execution times. Since the objective is optimizing execution time, the version of the algorithm that balances ILP and RP is used. For efficient compilation of CPU2006, the proposed B&B-SLIL algorithm was applied only to the hot functions. Compile times are reported in the next section.

Figures 8, 9, and 10 show the percentage improvements in execution speed achieved with the proposed B&B-SLIL algorithm relative to LLVM's default scheduling algorithm for the Xeon, i7 and ARM processors, respectively. For each processor, the graphs show only the benchmarks that are impacted by scheduling on that processor.

Our implementation of the proposed B&B-SLIL algorithm is a parametrized implementation that provides multiple options, including the choice of the heuristic for producing the initial feasible schedule (LLVM's heuristic or any derivative of the LUC heuristic described in our previous work (Shobaki et al. 2015)), instruction latencies (unit latencies, LLVM latencies or user-defined latencies), the time limit, the RPW in Equation (1), and the machine model.

Two machine models were used in the experiments: a simple machine model and an accurate machine model. The simple model assumes a single-issue processor that can issue an instruction of any type in any cycle but still captures latencies. The accurate model is based on LLVM's machine model and captures the issue rate, functional unit assignments and pipelining. In most cases, the overall results obtained with the accurate machine model were similar to those obtained with the simple machine model. This is attributed to the fact that the constraints imposed by the accurate machine model slow down the enumerative B&B search and limit the number of legal schedules that can be explored within a given time limit. Recall that for many large high-impact scheduling regions, the B&B scheduler does not converge to optimality within the time limit. We should note though that the accurate machine model gave significantly better results on a limited number of benchmarks, such as H264ref on ARM A7.

The blue bars in Figures 8, 9, and 10 show the improvements of the proposed B&B-SLIL algorithm relative to LLVM's algorithm with the same parameter settings used uniformly for all benchmarks. The parameter settings used in this test for each processor are shown in Table 6.

The orange bars in Figures 8 and 10 for Xeon and ARM A7 show the results obtained for each benchmark using the best parameter settings for that benchmark. No orange bar is displayed for Core i7, because using the settings in Table 6 gave the best results for all benchmarks. The settings that were varied for Xeon and ARM A7 included the initial heuristic (LLVM or LUC), the time limit (12 to 20ms/instruction for Xeon and 10 to 100ms/instr for ARM), instruction latencies (unit latencies or LLVM's latencies) and the RPW (0.008 to 0.03 on ARM only). The RPW weight was fixed at 1.0 for both Xeon and i7. Note that because the correlation between the scheduling cost function and execution time is not perfect, increasing the time limit does not always improve execution time.

(a) FP2006

(b) INT2006

Fig. 8. Percentage increase in execution speed relative to LLVM's default scheduler on Xeon.

The gray bar in each graph shows the percentage improvements when the CP heuristic is used for scheduling. Since CP is an ILP-only heuristic that tends to increase RP, the negative results of this heuristic measure the amount of performance degradation that results from exploiting ILP without taking RP into account. The simple machine model was used with CP.

The numbers in Figures 8, 9, and 10 are based on the median of three runs. If the random variation among the three runs (the difference between the highest score and the lowest score for the same benchmark using the same algorithm) was relatively significant, then zero improvement is shown in the graph. For example, if the difference in the median execution time between the B&B-SLIL algorithm and the base LLVM algorithm for a given benchmark was 1.2% and random variation among the three runs was 0.9%, a zero difference is shown in the graph for that benchmark, because of the low certainty about the measured difference.

On FP2006, the proposed algorithm gives geometric-mean improvements of 1.94% on Xeon, 4.22% on Core i7, and 5.12% on ARM A7. Double-digit percentage improvements are seen on Bwaves (23.75% on Xeon and 13.38 on i7), Lbm (49% on i7), and Namd (15.74% on ARM A7). Many other benchmarks show significant improvements. The proposed algorithm improves all six FP2006 benchmarks on ARM A7.

On INT2006, a significant geometric-mean improvement is seen only on ARM A7 with a maximum improvement of 8.21% on H264ref.

(a) FP2006

(b) INT2006

Fig. 9. Percentage increase in execution speed relative to LLVM's default scheduler on Core i7.

In the MediaBench results for ARM A7, the proposed algorithm improves the performance of 9 of 12 benchmarks. It does not improve g721+, jpeg+ and pegwit+, which are not shown in Figure 10, because they are not impacted by scheduling.

The results for the CP heuristic that tends to increase RP show that it results in substantial regressions on Cactus (34% on Xeon and 36% on i7) and Gromacs (13% on Xeon and 11% on i7). These two benchmarks have large basic blocks with high RP in their hot functions. So, it is not surprising that poor RP-aware scheduling of these hot blocks results in significant performance degradations. The results also show that the base algorithm, which is LLVM's default scheduling algorithm, is performing very well on these RP-sensitive benchmarks.

Examining the Core i7 results in Figure 9 shows that the proposed B&B-SLIL algorithm significantly outperforms the LLVM scheduler on many benchmarks with a maximum improvement of 49% on Lbm. A closer look at the improvements on Milc, Namd, Gems and Lbm indicates that these improvements must be ILP-related, because they are also produced by the CP heuristic. However, the improvement on Bwavs is likely to be RP-related, because it is not produced by CP and because it is seen on two different processors. The performance analysis results reported in Section 5.7 confirm that the improvement on Bwavs is indeed caused by a significant reduction in spilling. Overall, the execution-time results on Core i7 indicate that the proposed B&B-SLIL algorithm achieves a good balance between ILP and RP by producing all the ILP-related improvements produced by CP, avoiding all the RP-related regressions caused by CP and producing a significant RP-related improvement relative to LLVM on Bwavs.

(a) FP2006

(b) INT2006

(c) Mediabench

Fig. 10. Percentage increase in execution speed relative to LLVM's scheduler on ARM A7.

Comparing the ARM results in Figure 10 with the Intel results in Figures 8 and 9 shows that, because the ARM processor is in-order, the proposed B&B-SLIL algorithm consistently improves the performance of most benchmarks. The most noticeable difference between the ARM results and the Intel results is the significant improvements seen on the INT benchmarks for ARM.

Experimentally, achieving performance improvements on the in-order ARM processor was much more challenging than achieving performance improvements on the out-of-order Intel processors, because of the stronger need to balance ILP and RP. This balance was achieved by experimenting with the RPW parameter in Equation (1).

The proposed B&B-SLIL algorithm did not produce a significant improvement on every benchmark; some benchmarks were lightly impacted or not impacted at all. The limited improvement on some benchmarks is attributed to the following factors:

- (1) Improved scheduling can lead to a significant performance improvement only if it takes place in hot code (high-frequency basic blocks within hot functions). Register-pressure-aware scheduling can affect performance only if these hot basic blocks have high RP. The execution-time results in Figures 8, 9, and 10 show that at least five benchmarks (Bwaves, Gromacs, Cactus, Lbm and Namd) are highly sensitive to scheduling. Assuming that FP2006 is a representative sample of scientific programs, we expect 5 out of 17 scientific programs (29%) to be sensitive to scheduling, due to the need to balance RP with ILP. Twenty nine percent of all scientific programs is a large set of programs.
- (2) The optimality gaps in Figures 4, 5, and 6 show that although the proposed B&B-SLIL algorithm is statistically superior to LLVM's algorithm across all functions, it does not produce the minimum amount of spilling in every function. This is attributed to two factors. The first factor is that the correlation between the RP cost function and the amount of spill code is strong but not perfect. The second factor is that the proposed algorithm times out on thousands of scheduling regions. In future work, we will be exploring advanced algorithmic techniques to improve the algorithm's performance such that it solves significantly more scheduling regions to optimality.
- (3) Fewer benchmarks are impacted on Xeon and i7, because these are powerful out-of-order processors that have enough resources to absorb the cost of spill code. With powerful out-of-order execution, the compiler does not face the challenge of balancing ILP and RP; it can simply focus on minimizing RP. A particularly interesting target that we are considering for future work is a Graphics Processing Unit (GPU), because a GPU does not perform out-of-order execution and the cost of spilling is high (Rawat et al. 2018). Furthermore, reducing the number of registers used by each thread increases occupancy (the number of threads run in parallel). Therefore, it will be interesting to study the balance between ILP and RP on a GPU target.
- (4) Scheduling in LLVM is done locally within the basic block, and most basic blocks are small, especially in INT2006. Scheduling in a smaller basic block has less impact on performance.

5.5 Compile Times

Table 7 shows the total compile times of the FP2006 benchmarks on the Core i7 processor for both the base LLVM compiler and the modified compiler with the proposed B&B-SLIL applied only to the hot functions with a time limit of 10ms/instr. The compile times in the third column are the compile times for the test that produced the execution times in Figure 9.

Clearly, the proposed B&B-SLIL algorithm significantly increases the compile times of many benchmarks, with an overall increase of 86%. Although the increases in compile time are significant, our approach remains the closest to practicality among all published work on combinatorial

Table 7. Compile Times of FP2006
Benchmarks on Core i7 in Seconds

BENCHMARK	LLVM	B&B-SLIL
Bwaves	3	14
Gamess	122	135
Milc	5	10
Zeusmp	10	101
Gromacs	15	21
CactusADM	17	49
Leslie	6	36
Namd	7	53
DealII	73	74
Soplex	15	16
Povray	17	18
Calculix	31	41
GemsFDTD	11	76
Tonto	70	109
Lbm	2	5
Wrf	79	143
Sphinx3	5	7
Total	488	908 (86%)

instruction scheduling. Previously proposed combinatorial scheduling algorithms use time limits of minutes or even hours to schedule one function or one block (Barany and Krall 2013; Malik 2008; Winkel 2007). In contrast, the largest whole-program compile time reported in Table 7 is 143 seconds for Wrf, which is quite practical for such a large program. No other work on combinatorial instruction scheduling reported scheduling a whole program in seconds.

The compile times reported in the above table should be viewed as upper bounds, because:

- (1) Our current implementation is a research prototype with significant overhead. This overhead can be eliminated in a production version.
- (2) The hot-function selection may be revised to filter out the hot functions that are not impacted by scheduling.
- (3) We plan on exploring further algorithmic enhancements in future work.

5.6 Dependence on the Register Allocation Algorithm

In Section 5.3, we used the number of live ranges spilled by LLVM's greedy global register allocation algorithm (the default algorithm at higher optimization levels) as a metric for evaluating the relative performance of different scheduling algorithms. That comparison was based on the hypothesis that although the number of spills in any given function is highly dependent on the register allocation algorithm, the *total* spill count produced by *any* register algorithm across a large dataset is a valid metric for comparing different scheduling algorithms.

Register allocation is a complex problem that does not have a polynomial-time algorithm that produces the exact solution for every instance. Therefore, any register allocation algorithm will produce a mix of good solutions and bad solutions. Using a large dataset, however, is expected to give enough statistical significance to neutralize the dependence on the register allocation algorithm and provide an accurate method for comparing scheduling algorithms. To test this

Table 8. Spill-Code Statistics Produced by LLVM's Local Register Allocator in FP2006

	STAT	VALUE
1	Total functions	23,147
2	Functions with spills	18,533 (80%)
3	Sum of minima (SOM)	1,763,759
4	Avg. spills per function	76.2
5	Avg. spills per spilling function	95.2

ALGORITHM	EXTRA SPILLS	%FUNCS AT MIN	MAX EXTRA PER FUNC
BB_SLIL_RP	15,716 (0.89%)	79.3%	652
BB_PERP_RP	20,476 (1.16%)	74.2%	519
LLVM	23,235 (1.32%)	73.9%	654
BB_SLIL_RP&ILP	64,289 (3.64%)	76.5%	6969
BB_PERP_RP&ILP	67,609 (3.8%)	70.0%	6969
CP_ILP	108,975 (6.2%)	74.3%	5867

Optimality gaps for FP2006

Fig. 11. Algorithm ranking and optimality gaps in FP2006 produced by LLVM's local register allocator on Intel Xeon.

hypothesis, Table 8 shows the spill-code statistics and Figure 11 shows the optimality gaps using LLVM's local register allocation algorithm (used at lower optimization levels). As in the experiment reported in Figure 4, the algorithms under study were applied to all functions in FP2006.

Since finding the best possible SOM is not interesting in this test, only 9 algorithms were used to compute the SOM. Interestingly, the ranking of the algorithms in Figure 11 is the same as that in Figure 4, which confirms the hypothesis that the relative performance of scheduling algorithms is independent of the register allocation algorithm if the dataset is sufficiently large.

5.7 Case Studies

In this section, we take a closer look at the performance of some of the benchmarks that are highly impacted by scheduling

5.7.1 Bwaves. The proposed B&B-SLIL algorithm improves the execution time of Bwaves by 24% on Xeon. A 13% improvement is seen on Core i7. Bwaves has a dominant hot function, named `mat_times_vec_`, in which the program spends 71% of its execution time according to the profiling data that we have collected. The improvement in execution time is consistent with a substantial reduction in the *weighted spill count* in the dominant hot function. The weighted spill count in a given function is a weighted sum of the spill counts in the individual basic blocks, with the weight of each block's spill count being an estimate of the block's execution frequency. Using B&B-SLIL to schedule Bwaves's hot function reduces the weighted spill count from about 72 million to about

39 million. The high weighted spill counts reflect the fact that these spills are generated within a loop with a high trip count.

To confirm this, we have used the *Perf* tool to analyze the performance of Bwaves. *Perf* shows that applying the proposed scheduling algorithm causes a significant reduction in the number of memory accesses relative to LLVM's scheduling algorithm. The proposed algorithm reduces the L1-dcache load count from about 1,219 billion to about 1,007 billion (a 17% reduction) and the dynamic instruction count from about 3,068 billion to about 2,817 billion (an 8% reduction). These numbers show that the benchmark is memory-bound and that the spill instructions eliminated by the proposed B&B-SLIL algorithm constituted a significant percentage of the total number of instructions executed by the entire program.

5.7.2 Lbm. Lbm has one dominant hot function, in which the program spends 99% of its time. The proposed algorithm speeds up Lbm by 49% relative to the LLVM scheduler on Core i7. Our performance analysis shows that the cause of this improvement is a 58% reduction in memory stalls (from about 310 billion with LLVM to 131 billion with our scheduler). Interestingly, our algorithm increases the amount of spilling in this benchmark's hot function. So, the performance improvement is due to improved scheduling of memory operations not to a reduction in spilling. This test case shows that heavily biasing the scheduling algorithm toward RP reduction, as done in LLVM for Core i7, does not always give the best execution time. Balancing RP and ILP is necessary even on out-of-order processors. Our B&B-SLIL algorithm seems to achieve the right balance between RP and ILP in this case.

5.7.3 Cactus and Gromacs. According to the profiling data published by Weicker and Henning (2007), Cactus has a dominant hot function in which it spends 99.9% of its time, and Gromacs has a hot function in which it spends 66% of its time. Within the hot functions in Gromacs and Cactus, there are large basic blocks with hundreds of instructions. The scheduling of these basic blocks is quite challenging, due to both high RP and high ILP. LLVM's global register allocator produces substantial spilling in these hot functions. It produces over a thousand spills in Cactus's dominant hot function. We tried to schedule these regions optimally by increasing the time limit, but the B&B scheduler was not converging fast enough to produce significant reductions in spill code. We concluded that more advanced algorithmic techniques are needed to solve these challenging scheduling problems optimally, and that can potentially lead to significant performance improvements.

6 RELATED WORK

In our previous article on this problem (Shobaki et al. 2013), we provided a summary of the combinatorial techniques that were published prior to our work, including the work of Kessler (1998), Govindarajan et al. (2003), Malik (2008), and Barany and Krall (2013). We also covered the heuristic approaches proposed by Goodman and Hsu (1988), Berson et al. (1993), Touati (2005), Govindarajan et al. (2003), and Barany and Krall (2013). In this section, we cover the related articles that were published after the publication of our previous article.

Domagala et al. (2016) address the problem of minimizing RP using a combinatorial approach. However, the primary objective of their work is solving the loop unrolling problem. They present a Constraint Programming register-pressure-aware approach to integrating loop unrolling and instruction scheduling. Their solution also involves tiling. Loop unrolling and tiling are beyond the scope of our work.

Rawat et al. (2018) describe a source-level re-ordering algorithm to minimize register pressure for stencil computation on the GPU. Their approach is based on modeling stencil computation as a DAG of trees and then using the Sethi-Ullman algorithm (1970) to find the optimal order for

each tree. To schedule the DAG of trees, they use a greedy heuristic. Using this pattern-specific approach, they report speedups in the range of $1.22\times$ to $2.43\times$ for NVCC and $1.15\times$ to $2.08\times$ for LLVM. These results show the significance of register-pressure-aware scheduling on the GPU.

Lozano et al. (2018) recently present a combinatorial optimization approach to integrated instruction scheduling and register allocation using Constraint Programming. They present simulation-based performance results on in-order processors. Given the complexity of the integrated scheduling and allocation problem, their experimental results are quite promising. However, their approach is still impractical for production compilers; they use a time limit of 15 minutes per function on a powerful Intel processor with 12 hyper-threaded cores.

Finally, we explain the relation between our current article and our most related articles: Shobaki et al. (2013), referred to as TACO-2013, and Shobaki and Jamal (2015), referred to as COAP-2015. Our TACO-2013 article solves the same problem solved in the current article, where the objective is minimizing a weighted sum of schedule length and register pressure (compound objective), but the history domination technique in that article was limited to Case 1 of the history utilization algorithm described in Section 4.4. Our COAP-2015 article solves the SOP in which the objective is minimizing the sum of edge weights (simple objective), but the article describes a more general history utilization technique. In Section 4.4 of the current article, we describe how the general history utilization technique presented in the COAP-2015 article can be applied to the compound objective proposed in the TACO-2013 article.

7 CONCLUSIONS AND FUTURE WORK

This article explores the SLIL as an alternative cost function for combinatorial register-pressure-aware scheduling. A B&B algorithm is proposed for minimizing this cost function. The experimental results using a statistically significant dataset show that the proposed algorithm for minimizing SLIL gives significantly better results than the previous B&B algorithm for minimizing PERP.

Our analytical and experimental evaluations lead to the conclusion that it is unlikely to find a scheduling cost function that correlates perfectly with the amount of spill code generated by a separate register allocation algorithm. This confirms our previous conclusion that finding an instruction order that minimizes the amount of spill code generated by a given register allocation algorithm must be done by an instruction scheduling algorithm that is integrated within that register allocation algorithm (Shobaki et al. 2013). Integrated scheduling and allocation is the ultimate goal of this line of research.

In the short term, however, production compilers are likely to continue to perform scheduling and allocation in two different passes, due to the complexity of the integrated scheduling and allocation problem. Therefore, our short-term plan is to continue to improve our register-pressure-aware scheduling algorithm by exploring new algorithmic techniques for minimizing the proposed SLIL cost function and the PERP cost function. Such techniques include parallelization and graph transformations (Heffernan and Wilken 2005; Heffernan et al. 2006). Developing a register-pressure-aware scheduling algorithm for a GPU target is also an interesting topic for future work.

The experimental results in this article show that the proposed B&B-SLIL algorithm significantly improves the execution speeds of some CPU2006 benchmarks by up to 49%. The improved benchmarks are the benchmarks that have high RP in their hot code. Our experimental evaluation shows that 5 of the 17 FP2006 benchmarks are highly sensitive to scheduling. Assuming that FP2006 is a representative sample, we expect a large number of scientific programs to be highly sensitive to scheduling.

Our combinatorial approach to scheduling remains the most practical approach among all published work on combinatorial instruction scheduling. So far, it is the only approach that can compile a large program within seconds.

ACKNOWLEDGMENTS

The authors thank Lorenzo Smith, the Dean of ECS at CSUS for the additional funding that he provided to the second and third authors to complete the work after their graduation. The authors also thank Dibyendu Das, Shivarama Rao, Venugopal Raghavan and Prasad Vishwanath from the AMD CPU compiler team for providing the Dragon Egg version that was used in the implementation. The authors also thank Ted Krovetz for providing the ARM machine that was used in the experiments and Patrick Brannan for the technical support that he provided on the Intel machines. Thanks also go to Roberto Castaneda Lozano of the KTH Royal Institute of Technology in Sweden for the help that he provided to us on setting up the Mediabench benchmarks. We also thank LLVM developers Matthias Braun, Jonas Paulsson and Andrew Trick for answering our questions about the LLVM scheduler and register liveness information in LLVM. Finally, we thank the anonymous reviewers for their constructive comments and suggestions that led to a more thorough experimental evaluation in the final article.

REFERENCES

- G. Barany and A. Krall. 2013. Optimal and heuristic global code motion for minimal spilling. In *Proceedings of the International Conference on Compiler Construction*.
- D. Berson, R. Gupta, and M. Soffa. 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.
- T. Cormen, C. Leiserson, R. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.
- L. Domagala, D. Van Amstel, F. Rastello, and P. Sadayappan. 2016. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the International Conference on Compiler Construction*.
- J. Goodman and W. Hsu. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the International Conference on Supercomputing*.
- R. Govindarajan, H. Yang, J. Amaral, C. Zhang, and G. Gao. 2003. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Trans. Comput.* 52, 1 (2003), 4–20.
- M. Heffernan, K. Wilken, and G. Shobaki. 2006. Data-dependency graph transformations for superblock scheduling. In *Proceedings of the 39th International Symposium on Micro-architecture (MICRO 39)*.
- M. Heffernan and K. Wilken. 2005. Data-dependency graph transformations for instruction scheduling. *J. Schedul.* 8, 5 (2005), 427–451.
- C. Kessler. 1998. Scheduling expression DAGs for minimal register need. *Computer Languages* 24, 1 (1998), 33–53.
- M. Langevin and E. Cerny. 1996. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Des. Autom. Electr. Syst.* 1, 4 (1996), 443–456.
- C. Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. Media bench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*.
- R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte. 2018. Combinatorial register allocation and instruction scheduling. <https://arxiv.org/abs/1804.02452>.
- V. Makarov. 2013. Mechanism for Performing Instruction Scheduling based on Register Pressure Sensitivity. U.S. Patent No. 8,549,508.
- A. Malik. 2008. *Constraint Programming Techniques for Optimal Instruction Scheduling*. Ph.D. Thesis, University of Waterloo.
- P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L. Pouchet, A. Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’18)*.
- R. Sethi and J. D. Ullman. 1970. The generation of optimal code for arithmetic expressions. *J. ACM* 17, 4 (1970), 715–728.
- G. Shobaki, L. Sakka, N. Abu-Rmaileh, and H. Al-Hamash. 2015. Experimental evaluation of various register pressure reduction heuristics. *Softw. Pract. Exp.* 45, 11 (2015).
- G. Shobaki and J. Jamal. 2015. An exact algorithm for the sequential ordering problem and its application to switching energy minimization in compilers. *J. Comput. Opt. Appl.* 61, 2 (2015).

- G. Shobaki, M. Shawabkeh, and N. Abu-Rmaileh. 2013. Pre-allocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Trans. Arch. Code Optim.* 10, 3, Article 14 (2013).
- G. Shobaki. 2006. *Optimal Global Instruction Scheduling Using Enumeration*. Ph.D. dissertation, Department of Computer Science, UC Davis.
- G. Shobaki and K. Wilken. 2004. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th International Symposium on Microarchitecture*.
- SPEC CPU2006. 2006. Retrieved from <https://www.spec.org/cpu2006/>.
- S. Touati. 2005. Register saturation in instruction-level parallelism. *Intl. J. Parallel Prog.* 33, 4 (2005), 393–449.
- R. Weicker and J. Henning. 2007. Subroutine profiling results for the CPU2006 benchmarks. *ACM SIGARCH Computer Arch. News* 35, 1 (2007), 102–111.
- S. Winkel. 2007. Optimal versus heuristic global code scheduling. In *Proceedings of the 40th International Symposium on Microarchitecture*.

Received May 2018; revised October 2018; accepted November 2018