Graph Transformations for Register-Pressure-Aware Instruction Scheduling

Ghassan Shobaki
California State University, Sacramento
Sacramento, California, USA
ghassan.shobaki@csus.edu

Justin Bassett
California State University, Sacramento
Sacramento, California, USA
jbassett@csus.edu

Mark Heffernan
Google
USA
meheff@google.com

Austin Kerbow
California State University, Sacramento
Sacramento, California, USA
Austin.Kerbow@amd.com

Abstract

This paper presents graph transformation algorithms for register-pressure-aware instruction scheduling. The proposed transformations add edges to the data dependence graph (DDG) to eliminate solutions that are either redundant or sub-optimal. Register-pressure-aware instruction scheduling aims at balancing two conflicting objectives: maximizing instruction-level parallelism (ILP) and minimizing register pressure (RP). Graph transformations have been previously proposed for the problem of maximizing ILP without considering RP, which is a problem of limited practical value. In the current paper, we extend that work by proposing graph transformations for the RP minimization objective, which is an important objective in practice. Various cost functions are considered for representing RP, and we show that the proposed transformations preserve optimality with respect to each of them. The proposed transformations are used to reduce the size of the solution space before applying a Branch-and-Bound (B&B) algorithm that exhaustively searches for an optimal solution. The proposed transformations and the B&B algorithm were implemented in the LLVM compiler, and their performance was evaluated experimentally on a CPU target and a GPU target. The SPEC CPU2017 floating-point benchmarks were used on the CPU and the PlaidML benchmarks were used on the GPU. The results show that the proposed transformations significantly reduce the compile time while giving approximately the same execution-time performance.

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04.
https://doi.org/10.1145/3497776.3517771

CCS Concepts: • **Software and its engineering → Compilers**; • **Computing methodologies → Discrete space search**.

Keywords: instruction scheduling, register-pressure reduction, branch and bound, graph transformations, dominance

ACM Reference Format:
Ghassan Shobaki, Justin Bassett, Mark Heffernan, and Austin Kerbow. 2022. Graph Transformations for Register-Pressure-Aware Instruction Scheduling. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3497776.3517771

1 Introduction

Register allocation and instruction scheduling are two important and closely related compiler optimizations. The instruction order computed in the pre-allocation instruction scheduling pass determines register pressure (RP), which is the number of virtual registers with overlapping live ranges that cannot be assigned to the same physical register. If RP exceeds the number of physical registers on the target machine, the register allocator must spill some virtual registers to memory by adding load and store instructions (spill code) that may slow the program. Spilling is common in code generated for CPUs, especially compute-intensive code.

On a Graphics Processing Unit (GPU), spilling is rare and extremely expensive. However, RP determines the GPU occupancy, which is the number of thread groups that are executed in parallel. When each thread uses fewer registers, the GPU can run more threads in parallel. Occupancy usually has a high impact on the execution time of a GPU program.

In addition to minimizing RP, a pre-allocation instruction scheduling algorithm must exploit Instruction-Level Parallelism (ILP). ILP is exploited by executing more instructions in parallel to hide latencies, and thus minimize the schedule length. However, a higher degree of ILP tends to increase RP, as more registers are needed to hold the results of the instructions that are executed in parallel. Thus, maximizing

ILP and minimizing RP are two conflicting objectives that must be balanced in pre-allocation scheduling.

The problem of balancing ILP and RP in pre-allocation instruction scheduling is a fundamental problem in code generation and optimization. Optimizing either ILP or RP alone is NP-hard [4]. Current production compilers solve the RP-aware instruction scheduling problem using heuristics. However, recent research on both CPUs [18, 26] and GPUs [23, 25] has shown that these heuristics may produce sub-optimal schedules that significantly degrade performance.

To produce more precise solutions, combinatorial techniques have been recently proposed for instruction scheduling [1, 14, 18, 20, 25, 26, 28]. Although these techniques produce better schedules, they are much slower than the heuristics used in production compilers. Therefore, further algorithmic enhancements are needed to make combinatorial techniques fast enough for production compilers. In the current paper, we propose one such enhancement.

The proposed enhancement is a set of graph transformations (GTs) that modify the Data Dependence Graph (DDG) to reduce the size of the solution space before invoking a combinatorial algorithm. The proposed GTs reduce the solution space size by adding edges that eliminate redundant or sub-optimal solutions but still preserve at least one optimal solution. Such transformations are called *optimal transformations*. A transformation is optimal if every optimal solution to the modified DDG is also an optimal solution to the original DDG. In this paper, we present a set of sufficient conditions under which adding an edge between two instructions in the DDG is an optimal transformation. The GTs presented in this paper are based on *dominance* relations. A set of solutions A dominates a set of solutions B if the best solution in A is at least as good as the best solution in B.

In previous work, Heffernan and Wilken [10] described a DDG transformation algorithm that preserves ILP optimality, that is, the optimal schedule length for the transformed DDG is equal to the optimal schedule length for the original DDG. In our work, we present the register-pressure counterpart of the Heffernan-Wilken algorithm. Instead of proving optimality with respect to schedule length, we prove optimality with respect to RP, or more specifically, with respect to the RP cost functions defined in the paper.

The Heffernan-Wilken algorithm may be used only if RP can be ignored. In practice, RP can rarely be ignored, and that limits the practical value of those ILP-only GTs. On many target architectures, including GPUs and out-of-order CPUs with limited registers, minimizing RP is more important than minimizing schedule length [23, 25]. Therefore, the proposed RP-optimal GTs can play an important role in deploying combinatorial algorithms in production compilers.

On some target architectures, such as GPUs, both ILP and RP make a significant impact on performance. In this paper, we describe a straightforward way of combining the ILP-optimal transformations of Heffernan and Wilken and our proposed RP-optimal transformations to solve a formulation of the scheduling problem that balances ILP and RP.

The graph transformations proposed in the current paper are applied in a pre-processing step to the Branch-and-Bound (B&B) algorithm proposed by Shobaki et al. [25] for solving the RP-aware scheduling problem. This B&B algorithm is based on a two-pass formulation in which RP is minimized in the first pass, and schedule length is minimized in the second pass with the best RP found in the first pass used as a constraint. In the first pass, we pre-process the DDG using the proposed RP-optimal GTs, and in the second pass, we pre-process the DDG using a combination of our RP-optimal GTs and the ILP-optimal GTs of Heffernan and Wilken.

The proposed GTs and the B&B algorithm of Shobaki et al. were implemented in the LLVM compiler [16], and their performance was evaluated on an Intel x86 processor and an AMD GPU. The benchmarks used in the evaluation were the SPECspeed CPU2017 floating-point benchmarks [30] on the Intel target and the PlaidML benchmarks [13] on the AMD GPU target. The results show that the proposed GTs significantly reduce the compile time while still producing approximately the same execution time. On the Intel target, the proposed GTs reduce the number of instances that the B&B algorithm times out on by 45% and speedup the scheduling of the instances that do not time out by 84%. This leads to reducing the compile time by 13% while still producing approximately the same execution time.

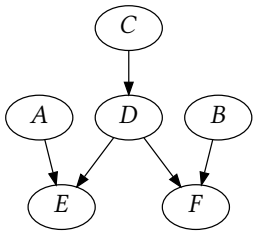
2 Background

The proposed GTs are used to solve the RP-aware instruction scheduling problem. The scope is limited to *local instruction scheduling*, that is, instruction scheduling within a *basic block*. A basic block is a straight-line piece of code with no branches out of it except at the end of the block and no branches into it except at the beginning of the block [4]. The input to the instruction scheduler is a data dependence graph (DDG), in which nodes represent instructions, edges represent dependencies, and edge weights represent latencies. The output is a schedule, which is an assignment of a machine cycle to each instruction in the input sequence.

An example DDG is shown in Figure 1a. The predecessor/successor relations among the DDG nodes play an important role in the proposed GTs. If there is an edge from Node x to Node y in a DDG, x is an *immediate predecessor* of y and y is an *immediate successor* of x. For a given node i in the DDG, IPred(i) is the set of immediate predecessors of i, excluding i itself, and ISucc(i) is the set of immediate successors of i, excluding i itself.

If there is a path (consisting of one or more edges) from node x to node y in a DDG, x is a *predecessor* of y and y is a *successor* of x. For a given node i in a DDG, Pred(i) is the set of predecessors of i, excluding i itself, and Succ(i) is the set of successors of i, excluding i itself. The predecessor and

(a)



(b)

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

(c)

Instr.	Pred(X)	IPred(X)	Succ(X)	ISucc(X)
A	$\emptyset$	$\emptyset$	$\{E\}$	$\{E\}$
B	$\emptyset$	$\emptyset$	$\{F\}$	$\{F\}$
C	$\emptyset$	$\emptyset$	$\{D, E, F\}$	$\{D\}$
D	$\{C\}$	$\{C\}$	$\{E, F\}$	$\{E, F\}$
E	$\{A, C, D\}$	$\{A, D\}$	$\emptyset$	$\emptyset$
F	$\{B, C, D\}$	$\{B, D\}$	$\emptyset$	$\emptyset$

(d)

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

(e)

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

Figure 1. (a) A DDG example from Shobaki et al. [26]. (b) Def and Use sets. (c) Predecessor and Successor sets. (d) First schedule with PRP = 3 and SLIL = 15. (e) Second schedule with PRP = 3 and SLIL = 13.

successor sets of each instruction in the DDG of Figure 1a are shown in Figure 1c.

Before applying the proposed GTs, we compute the *transitive closure* of the DDG and store in each node its predecessor/successor relation with other nodes in two different forms. The first form is a list of predecessors and a list of successors. The second form is two bit vectors for each node i that indicate if any other node j is a predecessor or a successor of i. If j is neither a predecessor nor a successor of i, then nodes i and j are *independent* instructions.

In RP-aware scheduling, the objective is finding a schedule that achieves the best possible balance between schedule length and RP. The schedule length is the number of cycles in the schedule, and RP is modeled using one of the cost functions described below.

The number of cycles in the schedule depends on the machine model. Our implementation of the proposed algorithm supports a general machine model. The experimental results, however, were produced using a simple machine model, in which the processor can issue one instruction of any type in each cycle. This simple model still captures instruction latencies, and, based on past experience, that appears to be the most important factor that affects performance.

In the pre-allocation scheduling phase, registers in the code are virtual registers. Each register has a specific data type. Register pressure computation is based on the Def and Use sets of the scheduled instructions. For a given instruction i, Def(i) is the set of registers that i defines, and Use(i) is the set of registers that i uses. Usually, RP is analyzed for each register type separately. For a given instruction i and a given register type T, $\text{Def}_T(i)$ is the set of registers of type T that i defines, and $\text{Use}_T(i)$ is the set of registers of type T that i uses. The Def and Use sets of each instruction in the DDG of Figure 1a are shown in Figure 1b.

An instruction can have an arbitrary number of Defs and Uses. It is also useful to track for each register R, Users(R), which is the set of instructions that use R.

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 if it has been defined, but at least one instruction that uses it has not been scheduled at that point.

In previous work, two different approaches have been used to solve the RP-aware scheduling problem: a single-pass approach [26, 28] and a two-pass approach [25]. In the single-pass approach, a weighted sum of schedule length and RP is optimized in a single pass.

On some targets, minimizing RP is much more important than minimizing schedule length. In theory, this can be captured in the single-pass approach by setting the weight of RP relative to schedule length to a sufficiently high value. Experimentally, however, it has been found that an extremely high RP weight results in a very slow algorithm [25].

That led to introducing a two-pass approach in which RP is minimized in the first pass as a primary objective and ILP is maximized in the second pass as a secondary objective [25]. In the first pass, ILP is totally ignored by setting all latencies to one. This allows the algorithm to focus on minimizing

RP in this pass. In the second pass, latencies are considered and the algorithm searches for a minimum-length schedule among all the schedules that maintain the best RP found in the first pass. In this work, we use the two-pass approach due to the importance of RP minimization on the target processors.

2.1 Register-Pressure Cost Functions

In previous work, multiple cost functions were proposed for representing RP during scheduling, including the *peak excess register pressure* (PERP) [28], the *sum of live interval lengths* (SLIL) [26], and the *adjusted peak register pressure* (APRP) [25]. These cost functions are defined next.

The *peak register pressure* (PRP) of a given data type in a given schedule is the maximum value of that type's RP at any point in the schedule. The PERP of a given data type is the difference between that type's PRP and the number of physical registers of that type on the target machine.

Assuming that the code is in Static Single Assignment (SSA) form [4], each virtual register in a basic block 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 the number of instructions in the instruction sequence that starts with the definition and ends with the last use. The SLIL is the sum of live interval lengths for all virtual registers in a given schedule. Since live interval overlapping makes live intervals longer, a larger SLIL indicates more overlapping among live intervals, and thus higher RP.

The DDG in Figure 1a consists of six 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 instructions E and F. The table in Figure 1b shows the Def and Use sets of each instruction.

The tables in Figure 1d and Figure 1e show two different schedules. The third column in each left-hand-side table shows the registers that are live at each point, and the fourth column shows the RP. The peak RP for both schedules is 3. If the target machine has 2 physical registers, the PERP will be 1 for both schedules.

The tables on the right sides of Figure 1d and Figure 1e show the calculation of the SLIL. The second column in each table shows the instructions that constitute the live interval for each register, and the third column shows the LIL. For example, the live interval for R_A in schedule 1 consists of Instructions A, B, D and E, and thus has a LIL of 4. The optimal LIL for R_A is 2 (A followed immediately by E).

Although both schedules have the same PERP, the SLIL is 15 for schedule 1 and 13 for schedule 2. This example shows that SLIL captures live interval overlaps that are not captured by PERP. SLIL captures the overlaps among all intervals, while PERP captures only the overlaps that lead to the peak pressure. In a high-pressure region, the peak-pressure point is not the only point at which the register allocator inserts spills. Therefore, minimizing SLIL is more likely to minimize spilling than minimizing PERP.

On a GPU, multiple PRP values may give the same occupancy. To model this, the adjusted peak register pressure (APRP) step function was introduced [25]. The APRP of a given PRP value x is the maximum PRP that gives the same occupancy as x. For example, on the AMD GPU used in this work, a PRP of 24 vector general-purpose registers (VGPRs) or less gives the maximum occupancy of 10 wavefronts, while PRP values in the range [25–28] give an occupancy of 9 wavefronts (a wavefront is a group of GPU threads that must be executed in lockstep). Therefore, PRP values in the range [1–24] are mapped to an APRP of 24 and PRP values in the range [25–28] are mapped to an APRP of 28.

3 Previous Work

Most previous work on instruction scheduling focused on scheduling for ILP, but some recent work addressed the RP-aware scheduling problem. Heuristic algorithms for balancing ILP and RP were proposed by Goodman and Hsu [8], Govindarajan et al. [9], Touati [31] and Barany and Krall [1].

In more recent work, some researchers proposed combinatorial approaches that compute exact solutions. Kessler [14] proposed a dynamic-programming solution. Barany and Krall [1] proposed an integer-programming solution. Malik [20] proposed a constraint-programming solution, and Domagala et al. [5] used constraint programming to integrate RP-aware scheduling and loop unrolling. Lozano et al. [17, 18] used constraint programming to solve the integrated scheduling and allocation problem. Lozano provides a survey of combinatorial approaches to instruction scheduling and register allocation [19].

Shobaki et al. [25, 26, 28] presented a B&B algorithm for solving the RP-aware instruction scheduling problem. B&B is an exact method for solving combinatorial optimization problems by conducting an exhaustive search of the solution space with some pruning techniques that make it possible to complete the search within reasonable time in most cases. In our work, we use the proposed GTs to pre-process the DDG before each pass of the two-pass scheduling algorithm proposed by Shobaki et al. [25].

Rawat et al. [23] describe an algorithm to minimize RP for stencil computation on the GPU. Their results show a significant impact of RP-aware scheduling on GPU performance.

The most related work on GTs for instruction scheduling in compilers is the work of Heffernan and Wilken [10, 11], who describe a GT algorithm that preserves ILP optimality. The algorithm presented in the current paper is the RP counterpart of the algorithm of Heffernan and Wilken. In an earlier paper, Wilken et al. describe various techniques for pre-processing the DDG before applying an integer-programming algorithm for solving the instruction scheduling problem [32].

The GT algorithm of Heffernan and Wilken is a generalization of the work of Ramamoorthy et al. [22] that was later extended by Chou and Chung [3]. The work of Ramamoorthy et al. is limited to zero-latency dependencies. Chou and Chung present a more general approach that accounts for latency constraints. The approach of Chou and Chung is based on a *generation tree* that enumerates possible solutions. Unlike our work, Chou and Chung do not use dominance relations to pre-process the DDG; they use them to prune inferior solutions within their enumerative search. Dominance relations for scheduling problems have also been proposed by Klein [15] and Dorndrof et al. [6].

Fernandez and Lang [7] describe an enumeration-based scheduler in which the DDG is transformed iteratively. Edges are added to the graph until all nodes are partitioned into completely ordered sets called *chains*.

Govindarajan et al. [9] propose a technique that adds edges to a DDG during scheduling to minimize register usage. The added edges create chains of instructions, called *lineages*, each of which is allocated a single register. Inagaki et al. [12] extend the technique of Govindarajan et al. and use that to minimizes register usage in a list scheduling framework with limited backtracking.

In addition to compiler instruction scheduling, graph transformations have been proposed for other combinatorial optimization problems, including the sequential ordering problem [21], the vehicle routing problem [2], and task scheduling in an operating system [29].

4 Algorithm Description

4.1 Register-Pressure Superiority

In this sub-section, we consider the problem of scheduling for the sole objective of minimizing RP. This is the problem solved in the first pass of the two-pass approach described above. An optimal transformation in this context is a transformation that preserves optimality with respect to each of the RP cost functions described above, namely PERP, SLIL, and APRP. More specifically, we seek a set of sufficient conditions under which we can insert an edge between two instructions such that an optimal solution to the modified DDG is also an optimal solution to original DDG.

To prove that adding an edge from Instruction x to Instruction y preserves both correctness and RP optimality, we must show that there exists at least one optimal schedule in which x appears before y. In this case, x is said to be *RP-superior* to y.

The approach that we take to proving superiority is based on a swapping argument. We first assume that there is an optimal schedule in which y appears before x and then show that swapping x and y will produce a correct schedule of cost less than or equal to the cost of the original schedule. This implies that for every optimal schedule in which y appears before x, there exists a corresponding optimal schedule in which x appears before y. Therefore, if we eliminate all the schedules in which y appears before x and limit our search to the schedules in which x appears before y, we will not miss an optimal schedule.

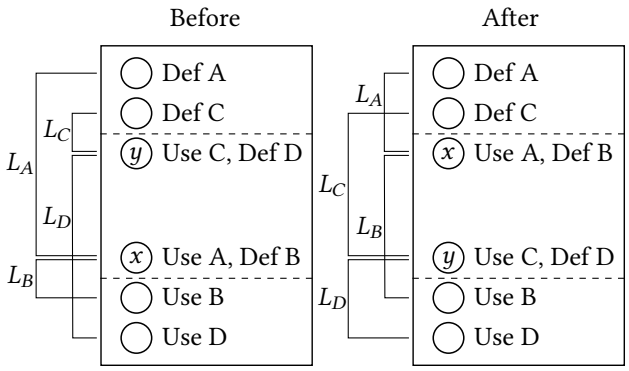


Figure 2. Schedules before and after swapping x and y.

To illustrate our swapping argument, consider the example in Figure 2, which shows two schedules. The first schedule is a general optimal schedule in which Instruction y appears before Instruction x, and the second schedule is the schedule that results from swapping x and y. The two instructions x and y are assumed to be of the same type. For example, both are integer arithmetic instructions.

For simplicity, we assume in this example that each instruction uses one register and defines another register. Specifically, Instruction x uses Register A and defines Register B, and Instruction y uses Register C and defines Register D. The graph shows the instructions that define A and C, and two instructions that use B and D. The graph also shows the live ranges of registers A, B, C, and D, assuming that every register use in the graph is the last use of that register.

First, we give a sufficient condition for the correctness (legality) of the swap. The swap moves x to an earlier position in the schedule. This will be legal if all the instructions that x depends on (Pred(x)) have been scheduled before that position. Since the original schedule is assumed to be legal, all the instructions that y depends on (Pred(y)) have been scheduled before that position. Therefore, if Pred(x) is a subset of Pred(y), moving x to y's original position will also be legal. With a similar argument about the set of instructions that depend on y (Succ(y)), we conclude that if Succ(y) is a subset of Succ(x), moving y to x's original position in the schedule will still be legal. Therefore, the swap will be legal under the following sufficient condition:

$$\text{Pred}(x) \subseteq \text{Pred}(y) \text{ and } \text{Succ}(y) \subseteq \text{Succ}(x)$$

Secondly, we give sufficient conditions for preserving RP optimality. The general idea is that the swap will extend (lengthen) some live ranges and reduce (shorten) other live ranges. As explained below, the amount of every live range extension or reduction will be exactly equal to the distance

between x and y in the schedule. Therefore, the swap is guaranteed to preserve optimality if, for every register type, the number of live ranges that *will* get extended is less than or equal to the number of live ranges that *may* get reduced.

It is important to note here that when counting live range extensions and reductions, we must count every possible extension but can count only guaranteed reductions. In other words, we must compute a lower bound (LB) on the number of reductions and an upper bound (UB) on the number of extensions. If the difference between the LB and the UB is non-negative, the swap will not increase RP at any point between x and y. If RP cannot increase at any point between x and y, none of the three cost functions (PERP, APRP, or SLIL) will increase after the swap.

We first consider the extensions and reductions that result from moving definitions. Assuming that the code is in SSA form, every live range has a unique definition. Moving x to an earlier position will extend the live range of every register that is defined by x. For example, see the extension of B's live range in Figure 2. On the other hand, moving y to a later position will reduce the live range of every register that is defined by y. For example, see the reduction of D's live range in Figure 2.

Unlike moving a definition, which always extends or reduces a live range (assuming SSA form), moving a use will not necessarily extend or reduce a live range. Moving a register use will change the register's live range length only if that use is the last use of that register. Since our objective is computing a LB on the number of reductions and an UB on the number of extensions, moving a register use to an earlier position cannot be counted as a reduction unless we prove that it is the last use. On the other hand, moving a register use to a later position must be counted as an extension unless we prove that it is not the last use.

Moving x to an earlier position will reduce the live range of a register that is used by x if x is the last user of that register. For example, in Figure 2, if x is the last user of Register A, the live range of A will get reduced after the swap. In general, x is necessarily the last user of Register R if every other instruction that uses R (if any) is in the predecessor list of x ($\text{Pred}(x)$). However, this can be relaxed further by allowing the instructions that use R to be in the predecessor list of y (recall that $\text{Pred}(x) \subseteq \text{Pred}(y)$). Since the only change that we make to the original schedule is swapping x and y, with all other instructions remaining in the same positions, a predecessor of y, which must appear before y in the original schedule, will appear before x after the swap, and thus x will still be the last user of R.

Therefore, to count the live ranges that will get reduced after the swap, we examine every register R in $\text{Use}(x)$, and then scan the list of instructions that use R. If every element in that list is in $\text{Pred}(y)$, that register's live range is counted as a reduced live range. The set of registers constructed for any register type T using this procedure is called UseShort_T. Formally, UseShort_T is defined as the set of registers R of type T such that for every instruction $i \in \text{Users}(R), i \in \text{Pred}(y) \cup \{x\}$.

Recall that before applying the GT algorithms, we compute the transitive closure of the DDG and store in each node i, among other things, two bit vectors that make it possible to determine in $O(1)$ time if any other node j is a predecessor or a successor of i.

We next consider the live range extension that may result from moving y to a later position. Moving y to a later position will extend the live range of a register R in $\text{Use}(y)$ if y is the last user of R. Therefore, the live range of every register R in $\text{Use}(y)$ must be counted as an extended live range unless we prove that y cannot be the last user of R. y cannot be the last user of R if at least one instruction that uses R is in $\text{Succ}(y)$. However, this can be relaxed further by allowing R to be used by an instruction in $\text{Succ}(x)$. Again, since we are swapping x and y while keeping all other instructions in their positions, a successor i of x, which must appear after x in the original schedule, must appear after y after the swap. Thus, if i uses R, y cannot be the last user of R.

Therefore, to count the live ranges that may get extended after the swap, we examine every register in $\text{Use}(y)$, and for each register, we scan the list of instructions that use it. If at least one element in that list is in $\text{Succ}(x)$, that register's live range is not counted as an extended live range. The set of register live ranges that is constructed using this procedure for any register type T is called UseLong_T. Formally, UseLong_T is defined as the set of registers R of type T satisfying:

1. $R \in \text{Use}_T(y)$,
2. $R \notin \text{Use}_T(x)$,
3. For every instruction $i \in \text{Users}(R)$, $i \notin \text{Succ}(x)$.

Putting all the above ideas together leads to the following theorem.

Theorem 1. *Given a DDG, adding a zero-latency edge from x to y in that DDG is a register-pressure optimal transformation if nodes x and y satisfy the following conditions:*

1. *Nodes x and y are independent nodes in the DDG,*
2. $\text{Type}(x) = \text{Type}(y)$,
3. $\text{Pred}(x) \subseteq \text{Pred}(y)$,
4. $\text{Succ}(y) \subseteq \text{Succ}(x)$,
5. *For each register type T,*

$$\text{NumLong}_T \leq \text{NumShort}_T$$

where
$\text{NumLong}_T = |\text{UseLong}_T| + |\text{Def}_T(x)|$
$\text{NumShort}_T = |\text{Def}_T(y)| + |\text{UseShort}_T|$

The proof of Theorem 1 is omitted for space limitations.

4.2 Algorithm

Based on the above theorem, we have developed the following algorithm for checking RP superiority and adding edges to the DDG.

```
1:  procedure RP-SUPERIOR-NODE-ALGORITHM(G)
2:    for all instruction A ∈ V(G) do
3:      for all instruction B ∈ V(G) do
4:        if A and B are independent then
5:          if IS-RP-SUPERIOR(A, B) then
6:            Add latency-zero edge (A, B) to E(G)
7:            REMOVE-REDUNDANT-EDGES(G, A, B)
8:          else if IS-RP-SUPERIOR(B, A) then
9:            Add latency-zero edge (B, A) to E(G)
10:           REMOVE-REDUNDANT-EDGES(G, B, A)
11: procedure IS-RP-SUPERIOR(A, B)
12:   if Type(A) ≠ Type(B) then return false
13:   if Pred(A) ⊈ Pred(B) then return false
14:   if Succ(B) ⊈ Succ(A) then return false
15:   LENGTHENED-COUNT ← Array of zeros with length number-of-register-types.
16:   for all register R ∈ Use(B) do
17:     if R ∉ Use(A) and ∀C ∈ Users(R) − {A, B}, C ∉ Succ(A) then
18:       increment LENGTHENED-COUNT(Type(R))
19:   for all register R ∈ Use(A) do
20:     if ∀N ∈ Users(R), N ∈ Pred(B) ∪ {A} and N ≠ B then
21:       decrement LENGTHENED-COUNT(Type(R))
22:   for all register R ∈ Def(A) do
23:     increment LENGTHENED-COUNT(Type(R))
24:   for all register R ∈ Def(B) do
25:     decrement LENGTHENED-COUNT(Type(R))
26:   for all C ∈ LENGTHENED-COUNT do
27:     if C > 0 then return false
28:   return true
29: procedure REMOVE-REDUNDANT-EDGES(G, A, B)
30:   for all node P ∈ Pred(A) ∪ {A} do
31:     for all node S ∈ ISucc(P) do
32:       if (P, S) ≠ (A, B) and S ∈ Succ(B) ∪ {B} then
33:         remove (A, B) from E(G)
```

4.3 Combining RP and ILP

As mentioned above, the approach used in this work is a two-pass approach. In the first pass (the RP pass), the RP-only conditions described in the previous sub-sections are applied, because only RP is considered in this pass. In the second pass (the ILP pass), however, both RP and ILP are considered. Therefore, the conditions for adding edges to the DDG must preserve both RP optimality and ILP optimality. Such conditions can be derived by simply combining the RP-only conditions proposed in the current paper with the ILP-only conditions described by Heffernan and Wilken [10]. In other words, an edge can be added from Node x to Node y in the DDG if x and y satisfy *both* the RP superiority conditions stated in Theorem 1 and the ILP superiority conditions given by Heffernan and Wilken. This implies that the edges added in the second pass (the ILP pass) will be fewer than the edges added in the first pass, because in the second pass, two sets of conditions must be satisfied, while in the first pass, only one set of conditions must be satisfied.

For example, using the setup described in Section 5 for the SLIL cost function, 2,131,387 edges were added in the first pass. In the second pass, 627,715 edges satisfied the ILP conditions of Heffernan and Wilken, but only 315,221 (50.2%) edges satisfied both the ILP conditions and the RP conditions.

4.4 Integrating Graph Transformations with B&B

The proposed GT algorithms were implemented and used to preprocess the DDG before it is input to the two-pass B&B scheduling algorithm described by Shobaki et al. [25]. In the first pass, each scheduling region is first scheduled using a heuristic, and then a LB on the RP cost is computed. If the heuristic cost is equal to the LB, the schedule is already RP optimal. Otherwise, the region is passed to the B&B algorithm to search for an optimal schedule within a certain time limit. If no optimal schedule is found within the time limit, the best schedule found is taken.

In the first pass, only the RP GTs are applied after constructing the heuristic schedule if it is not optimal. Since GTs often cause the LB algorithm to compute a tighter LB, the LB is recomputed after applying the GTs. If the recomputed LB is equal to the cost of the heuristic schedule, the heuristic schedule is optimal, and B&B is not invoked. This is an added benefit of applying GTs. If optimality is not proven, the modified DDG is passed to the B&B scheduler.

The best schedule found in the first pass, whether it is optimal or not, is input into the second pass, after inserting enough stalls to satisfy the latency constraints. In the second pass, the B&B algorithm searches for the shortest schedule that maintains the best RP found in the first pass. The combined RP-ILP graph transformations are applied to the DDG immediately after setting up the input schedule.

5 Experimental Results

5.1 Experimental Setup

The proposed GT algorithm was implemented together with the two-pass B&B scheduling algorithm proposed by Shobaki et al. [25] in the LLVM compiler, and the *modified LLVM compiler* was applied to a CPU target and a GPU target.

The CPU target is an Intel Core i7-7700K processor running at 4.2GHz. For this target, LLVM 7.1.0 was used, and the

Table 1. Benchmark statistics

	Stat	FP2017	Plaidbench
1	Number of benchmarks	8	13
2	Number of functions/kernels	10 233	3 814
3	Number of scheduling regions	464 970	16 682
4	Avg. scheduling regions per function/kernel	45.44	4.37
5	Avg. Instructions per scheduling region	7.88	47.02
6	Max. Instructions per scheduling region	5 197	919

evaluation was done using the floating-point benchmarks in SPECspeed[1] CPU 2017 (FP2017 for short) [30].

The GPU target is an AMD Radeon RX Vega 64 GPU running at 1.63 GHz. For this target, we used the LLVM in rococl-2.4.0, and the evaluation was done using the PlaidML benchmarks [13]. Compilation was done on an AMD Ryzen Threadripper 1950X processor.

On the Intel target, a time limit of 5ms/instruction was used in each pass of the B&B algorithm. For example, a scheduling region with 100 instructions was given a limit of 500ms. On the AMD GPU target, the time limit was 20ms/instruction in the first pass and 2ms/instruction in the second pass. The initial heuristic schedule was computed using the Last Use Count (LUC) heuristic [27]. The `-O3` optimization level was used in all tests. Table 1 shows interesting statistics about the benchmarks used in the experiments.

5.2 Effect of Graph Transformations

To evaluate the effectiveness of the proposed GTs, the benchmarks were compiled using the modified LLVM compiler with and without GTs. For the Intel target, the SLIL and the PERP cost functions were used. Table 2 shows the statistics for SLIL and Table 3 shows the same statistics for PERP.

First, we comment on the results in Table 2. Table 2a shows the low-level statistics in each pass individually, and Table 2b shows the high-level statistics including both passes. For each pass, the first column shows the value of the stat when GTs are disabled, the second column shows the corresponding value when GTs are enabled, and the third column shows the percentage improvement from applying GTs.

In the first pass, Row 1 shows that with GTs, 18% fewer scheduling regions were passed to B&B, because applying GTs resulted in tighter LBs. Row 2 shows that applying GTs reduced the number of regions that timed out from 8734 to 4805, a 45% reduction. Row 3 shows that using GTs reduced the solution time for the scheduling regions that were solved only with GTs enabled by 78%. Rows 4 and 5 show that applying GTs greatly sped up the scheduling of the regions that were solved optimally with and without GTs. Row 5 shows that GTs reduced the number of tree nodes enumerated by the B&B algorithm in scheduling these regions by 86%, and Row 4 shows that this sped up the B&B search by 84%.

Rows 6 and 7 show that these improvements reduced the B&B time by 30% and the total scheduling time in the first pass by 29%. Row 8 shows that using GTs did not only reduce the number of timeouts and speedup the search, but it also reduced the overall SLIL cost across all regions by 4%.

The right-hand-side of Table 2a shows the same statistics collected in the second pass. Recall that in the second pass, we apply the combined RP and ILP GTs, which are much more restrictive than the RP-only GTs. In the combined GTs, an edge is added if it is both RP-optimal and ILP-optimal. This explains why the percentage improvements achieved by applying GTs in this pass are generally smaller than the corresponding percentage improvements in the first pass. However, the percentage improvements in the second pass are still substantial.

Table 2b shows the high-level statistics across both passes. These statistics show that the overall effect of applying GTs in both passes is reducing the total scheduling time by 16%, which leads to reducing the total compile time by 13%. Row 3 shows that applying GTs reduces the total number of spills by 0.6% but increases the total schedule length by 0.3%. The slight increase in schedule length is attributed to the fact that when the B&B algorithm finds a schedule with a lower RP in the first pass, that imposes a stronger constraint on the search for a shorter schedule in the second pass.

It is noted that with the use of B&B scheduling, the total compile time is dominated by the scheduling time. This suggests that in practice, B&B should be selectively applied to the hot regions not to all regions. B&B scheduling is applied to all regions in this subsection only to maximize statistical significance. The experiments of the next subsection are based on the more practical choice of applying B&B only to the hot scheduling regions in FP2017.

Table 3 shows that similar significant improvements were obtained for PERP. Note that the difference in Row 1 is zero, because there is currently no known lower-bound algorithm for PERP. It is also noted that applying GTs in the second pass changed the search in a way that took 1.6% more time but resulted in a total schedule length that is 3% better.

Table 4 shows the results for the PlaidML benchmarks on the AMD GPU. On PlaidML, GTs were applied only to the first pass, as they had limited effect on the second pass. Table 4a shows the stats for the first pass. A total of 340 regions were passed to B&B. Row 2 shows that GTs reduced the number of timeouts from 203 to 165, an 18% reduction. Row 3 shows that, in addition to reducing the number of timeouts, GTs resulted in a 9% reduction in the solution time of the instances that were solved optimally with and without GT. Row 4 shows that this reduction in time is consistent with approximately the same percentage reduction in the number of tree nodes explored. The overall effect of all the above reductions is a 2.1% reduction in B&B time (Row 5) and a 1.9% in the total scheduling time in the first pass. Row 7 shows that, in addition to reducing compile time, solving more instances to optimality reduced the APRP cost by 11%.

Table 4b shows that the reductions in APRP improved occupancy on 40 kernels. For 39 kernels, the occupancy increased from 8 to 9, and for one other kernel, it increased from 2 to 3. There is one regression from 10 to 9 on one kernel. Regressions in occupancy are unavoidable, because the correlation between APRP and occupancy is strong but

[1] wrf and pop2 were excluded due to runtime errors that were seen even with the base LLVM compiler. imagick was excluded from the execution-time results due to a large regression when GTs are disabled. When GTs are enabled, this regression was not present, but the reason behind this regression is believed to be unrelated to GTs or to scheduling at all.

Table 2. Stats for FP2017 using SLIL. (a) Low-level stats. (b) High-level stats across both passes.

(a)

	Stat	First pass			Second pass		
		Without GT	With GT	% imp.	Without GT	With GT	% imp.
1	Number of regions passed to B&B	231 077	188 407	18	91 189	87 681	4
2	Number of regions timed out	8 734	4 805	45	13 999	12 540	10
3	Solution time for GT-only-opt (s)	537	120	78	172	67	61
4	B&B time for both-opt (s)	335	54	84	194	107	45
5	Tree nodes for both-opt	199 165 481	27 010 889	86	103 979 881	50 507 289	51
6	Total B&B time (s)	2 292	1 602	30	2 800	2 607	7
7	Total scheduling time in this pass (s)	2 386	1 701	29	2 912	2 743	6
8	Total relative RP cost	8 693 209 000	8 347 881 000	4	–	–	–
9	Total schedule length	–	–	–	602 429	598 058	0.7

(b)

	Stat	Without GT	With GT	% imp.
1	Total compile time (s)	7 055	6 129	13
2	Total scheduling time in both passes (s)	5 484	4 622	16
3	Total number of spills	155 863	154 913	0.6
4	Total schedule length	686 945	689 337	-0.3

Table 3. Stats for FP2017 using PERP. (a) Low-level stats. (b) High-level stats across both passes.

(a)

	Stat	First pass			Second pass		
		Without GT	With GT	% imp.	Without GT	With GT	% imp.
1	Number of regions passed to B&B	4 164	4 164	0	79 337	79 337	0
2	Number of regions timed out	2 180	861	61	366	276	25
3	Solution time for GT-only-opt (s)	236	39	84	25	14	46
4	B&B time for both-opt (s)	37	3	91	21	16	25
5	Tree nodes for both-opt	55 552 021	3 755 061	93	5 863 493	4 146 002	29
6	Total B&B time (s)	894	661	26	359	343	5
7	Total scheduling time in this pass (s)	965	736	24	456	464	-1.6
8	Total relative RP cost	115 840 000	115 792 000	0.04	–	–	–
9	Total schedule length	–	–	–	104 930	101 688	3

(b)

	Stat	Without GT	With GT	% imp.
1	Total compile time (s)	2 539	2 259	11
2	Total scheduling time in both passes (s)	1 468	1 213	17
3	Total number of spills	163 448	163 527	-0.05
4	Total schedule length	128 663	125 908	2

not perfect. APRP is the scheduler's estimate of RP, while the occupancy depends on the actual number of registers used, which is determined by the register allocation algorithm. Since register allocation is NP-hard, the register allocation algorithm may produce sub-optimal results in some cases.

5.3 Execution Times

For the execution-time tests on FP2017, B&B scheduling was applied only to the hot functions. Before we present the execution-time results, we show in Table 5 the high-level statistics for the hot scheduling regions using SLIL. Rows 1 and 2 show that GTs reduced the total compile time by 1.0% and the total scheduling time by 1.2%. The total number

Table 4. Stats for Plaidbench. (a) Low-level stats for the first pass. (b) Number of kernels with altered occupancies.

(a)

	Stat	Without GT	With GT	% imp.
1	Number of regions passed to B&B	340	340	0
2	Number of regions timed out	203	165	18
3	B&B time for both-opt (s)	132	121	9
4	Tree nodes for both-opt	129 398 534	116 204 157	10
5	Total B&B time (s)	1 178	1 154	2.1
6	Total scheduling time in this pass (s)	1 180	1 157	1.9
7	Total relative RP cost	33 080	29 340	11

(b)

	Change in occupancy		
	Without GT	With GT	Count
1	8	9	39
2	2	3	1
3	10	9	1

Table 5. Stats for FP2017 using SLIL on hot functions.

	Totalled stat	Without GT	With GT	% imp.
1	Compile time (s)	1 336	1 323	1.0
2	Scheduling time in both passes (s)	810	800	1.2
3	Number of spills	19 842	19 842	0
4	Weighted spills	2.023e9	1.858e9	8
5	Schedule length	102 681	102 866	-0.2

Table 6. Percentage gain in FP2017 speed relative to LLVM.

	Benchmark	B&B without GT	B&B with GT
1	603.bwaves_s	0.8%	0.8%
2	607.cactuBSSN_s	2.5%	3.2%
3	619.lbm_s	4.3%	4.5%
4	627.cam4_s	0.2%	0.6%
5	644.nab_s	-0.2%	0.4%
6	649.fotonik3d_s	0.8%	2.1%
7	654.roms_s	-0.3%	0.1%
8	Geo-mean	1.2%	1.7%

of spills is the same, but the total weighted spill count is reduced by 8%. The weighted spill count is a weighted sum that accounts for the statically estimated execution frequency of each spill. In Table 6, it is shown that this 8% reduction in the weighted spill count results in a slight improvement of the execution time when GTs are enabled.

Table 6 shows the percentage gain in FP2017 execution speed using B&B scheduling with and without GTs relative to LLVM's generic scheduler. Table 7 shows the percentage gains in PlaidML execution speed relative to AMD's production scheduler. Each test was run five times, and the percentage gains in the table are based on the median scores.

It is noted that the performance gains achieved using a B&B scheduler on the AMD GPU are significantly greater than the gains achieved on the Intel CPU. On the AMD GPU, a geometric-mean improvement of 8% is achieved, while the geometric-mean improvement on Intel is 1.2% without GTs and 1.7% with GTs. This is attributed to the fact that both RP and ILP have a higher impact on GPU performance than on CPU performance. On the GPU, RP determines occupancy, while on the CPU it affects spilling. The impact of occupancy on GPU performance is generally higher than the impact of spilling on CPU performance. Spilling significantly affects performance only if there is a substantial amount of spilling in hot code. Furthermore, compiler scheduling for ILP has a higher impact on GPU performance, because the GPU is in-order, while the Intel processor is out-of-order.

Table 7. Percentage gain in PlaidML speed relative to AMD.

	Benchmark	B&B without GT	B&B with GT
1	densenet121	4.5%	4.4%
2	densenet169	5.0%	5.0%
3	densenet201	4.5%	4.6%
4	inception_resnet_v2	19.5%	19.2%
5	inception_v3	18.8%	18.7%
6	mobilenet	4.8%	4.8%
7	nasnet_large	8.3%	8.3%
8	nasnet_mobile	7.4%	7.4%
9	resnet50	7.8%	6.9%
10	vgg16	9.4%	9.4%
11	vgg19	8.9%	8.9%
12	xception	6.6%	6.6%
13	imdb_lstm	1.8%	1.9%
14	Geo-mean	8.1%	8.0%

The results in the previous section show that applying GTs significantly reduces compile time. The results in Tables 6 and 7 show that this reduction in compile time is achieved without degrading the execution-time performance. In fact, on FP2017, the execution time is slightly better with GTs.

Table 8. Hot scheduling regions in Cactus with a time limit of 220 ms/instruction in the first pass.

	Stat	Without GT	With GT	% imp.
1	Regions passed to B&B	326	326	0
2	Regions timed out	41	36	12
3	Solution time for GT-only-opt (ms)	34 545	40	99.9
4	Tree nodes for both-opt	6 074 425	1 033 118	83
5	B&B time for both-opt (ms)	9 198	1 704	81
6	Total spill count	6 605	6 509	1

5.4 Longer Time Limits

To shed more light on the performance of the proposed GTs, a study was performed on the Cactus benchmark using a significantly longer time limit. Cactus was chosen, because the hot functions in Cactus have many hard-to-schedule regions with extremely high register pressure. The register allocator generates thousands of spills in Cactus's hot functions.

Cactus was compiled using the modified compiler with B&B scheduling applied only to the hot functions with a time limit of 220 ms/instruction in the first pass. Table 8 shows the statistics of this experiment with and without GTs. Row 1 shows that the hot functions in Cactus have a total of 326 scheduling regions. Row 2 shows that applying GTs resulted in optimally scheduling five regions that timed out without GTs. Row 3 shows that the total scheduling time for these five regions is 34 545 ms without GTs and 40 ms with GTs, a 99.9% reduction. Rows 4 and 5 show that for the regions that were scheduled optimally with and without GTs, applying GTs reduced the number of enumerated tree nodes by 83% and the B&B time by 81%. Row 6 shows that using GTs resulted in reducing the number of spills by 96.

Table 9 shows more details about the five scheduling regions that timed out without GTs but were solved to optimality with GTs. The scheduling times for these regions are thousands of milliseconds without GTs. Applying GTs reduces the scheduling time to a few milliseconds for four regions and to 29 ms for the fifth region. Solving these five regions to optimality led to discovering zero-cost schedules for four of them. These results show that GTs can potentially have a high impact on the B&B scheduling of some regions.

The authors believe that the proposed GTs had a high positive impact on some but not all hard scheduling regions, because for many hard regions, the B&B scheduler may get trapped in a solution sub-space that does not have better schedules. Such trapping may happen if bad decisions are made in the earlier steps of the exploration (at the shallower nodes in the tree). The solution to such a problem is the parallelization of the B&B search, as that makes it possible to explore multiple sub-spaces in parallel and focus the search on the most promising sub-spaces.

Table 9. Hot scheduling regions in Cactus solved with GTs.

	Region Size	Solution time		SLIL cost	
		Without GT (ms)	With GT (ms)	Without GT	With GT
1	32	6 613	3	21 000	0
2	30	6 170	3	17 000	0
3	60	12 798	29	178 000	177 000
4	21	4 201	4	11 000	0
5	24	4 851	2	5 000	0

We are currently working on parallelizing the B&B search. We believe that a combination of parallelization and GTs can solve many of the currently unsolvable instances of the scheduling problem.

6 Conclusions and Future Work

In this paper, we present RP-optimal graph transformations that reduce the size of the solution space of a B&B algorithm for solving the RP-aware instruction scheduling problem. The proposed transformations reduce the number of time-outs by 45% and speed up the scheduling of the instances that do not time out by 84%.

For future work, we plan on exploring a dynamic version of the proposed transformations as a pruning technique in B&B. A dynamic version can potentially relax the restrictive conditions for the combined RP and ILP GTs. Additionally, we plan on investigating an iterative version of the RP GTs, wherein added edges can enable the addition of more edges. Furthermore, we plan on using the proposed GTs with a parallel version of the B&B algorithm and with an Ant Colony Optimization algorithm that we have recently developed for solving the RP-aware scheduling problem [24].

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) through Award No. 1911235. The authors thank the GPU-compute compiler team at Advanced Micro Devices (AMD) for providing technical consultation and donating the machine that was used to generate the GPU results in this paper. The authors also thank Patrick Brannan and Lynne Koropp for the technical support that they provided and Vang Thao for the help that he provided on the experimental setup. Finally, we thank the anonymous reviewers for their constructive comments and suggestions that led to improving the final paper.

References

[1] Gergö Barany and Andreas Krall. 2013. Optimal and Heuristic Global Code Motion for Minimal Spilling. In *Proceedings of the 22nd International Conference on Compiler Construction* (Rome, Italy) *(CC'13)*. Springer-Verlag, Berlin, Heidelberg, 21–40. https://doi.org/10.1007/978-3-642-37051-9_2

[2] Beck J.C., Prosser P., Selensky E. 2002. Graph Transformations for the Vehicle Routing and Job Shop Scheduling Problems. In *Lecture Notes in Computer Science (ICGT 2002, Vol. 2505)*, Corradini A., Ehrig H., Kreowski H.J., Rozenberg G. (Ed.). Springer, Berlin, Heidelberg, 60–74. https://doi.org/10.1007/3-540-45832-8_7

[3] Hong-Chich Chou and Chung-Ping Chung. 1995. An Optimal Instruction Scheduler for Superscalar Processor. *IEEE Trans. Parallel Distrib. Syst.* 6, 3 (mar 1995), 303–313. https://doi.org/10.1109/71.372778

[4] Keith Cooper and Linda Torczon. 2011. *Engineering a Compiler* (2 ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[5] Łukasz Domagała, Duco van Amstel, Fabrice Rastello, and P. Sadayappan. 2016. Register Allocation and Promotion through Combined Instruction Scheduling and Loop Unrolling. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 143–151. https://doi.org/10.1145/2892208.2892219

[6] Ulrich Dorndorf, Toàn Phan Huy, and Erwin Pesch. 1999. A survey of interval capacity consistency tests for time-and resource-constrained scheduling. In *Project Scheduling*. Springer, 213–238. https://doi.org/10.1007/978-1-4615-5533-9_10

[7] E. B. Fernandez and T. Lang. 1976. Scheduling as a Graph Transformation. *IBM Journal of Research and Development* 20, 6 (1976), 551–559. https://doi.org/10.1147/rd.206.0551

[8] J. R. Goodman and W.-C. Hsu. 1988. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 2nd International Conference on Supercomputing* (St. Malo, France) *(ICS '88)*. Association for Computing Machinery, New York, NY, USA, 442–452. https://doi.org/10.1145/55364.55407

[9] R. Govindarajan, Hongbo Yang, José Nelson Amaral, Chihong Zhang, and Guang R. Gao. 2003. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures. *IEEE Trans. Comput.* 52, 1 (Jan. 2003), 4–20. https://doi.org/10.1109/TC.2003.1159750

[10] Mark Heffernan and Kent Wilken. 2005. Data-Dependency Graph Transformations for Instruction Scheduling. *J. of Scheduling* 8, 5 (Oct. 2005), 427–451. https://doi.org/10.1007/s10951-005-2862-8

[11] Mark Heffernan, Kent Wilken, and Ghassan Shobaki. 2006. Data-Dependency Graph Transformations for Superblock Scheduling. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 77–88. https://doi.org/10.1109/MICRO.2006.16

[12] Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. 2003. Integrated Prepass Scheduling for a Java Just-In-Time Compiler on the IA-64 Architecture. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 159–168.

[13] Intel. 2017. PlaidML machine learning benchmarks. https://github.com/plaidml/plaidbench#intel-corporation-machine-learning-benchmarks

[14] Christoph W. Kessler. 1998. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages* 24, 1 (1998), 33–53. https://doi.org/10.1016/S0096-0551(98)00002-2

[15] Robert Klein. 2001. *Scheduling of Resource Constrained Projects*. Kluwer Academic Publishers, USA.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

[17] Roberto Castañeda Lozano. 2018. *Constraint-Based Register Allocation and Instruction Scheduling*. Ph.D. Dissertation. KTH Royal Institute of Technology.

[18] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. on Programming Languages and Systems* 41, 3, Article 17 (July 2019), 53 pages. https://doi.org/10.1145/3332373

[19] Roberto Castañeda Lozano and Christian Schulte. 2019. Survey on Combinatorial Register Allocation and Instruction Scheduling. *Comput. Surveys* 52, 3, Article 62 (June 2019), 50 pages. https://doi.org/10.1145/3200920

[20] Abid Malik. 2008. *Constraint Programming Techniques for Optimal Instruction Scheduling*. Ph.D. Dissertation. University of Waterloo.

[21] R. Montemanni, D. H. Smith, and L. M. Gambardella. 2008. A Heuristic Manipulation Technique for the Sequential Ordering Problem. *Comput. Oper. Res.* 35, 12 (dec 2008), 3931–3944. https://doi.org/10.1016/j.cor.2007.05.003

[22] C. V. Ramamoorthy, K. M. Chandy, and Mario J. Gonzalez. 1972. Optimal Scheduling Strategies in a Multiprocessor System. *IEEE Trans. Comput.* C-21, 2 (1972), 137–146. https://doi.org/10.1109/TC.1972.5008918

[23] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas 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* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 168–182. https://doi.org/10.1145/3178487.3178500

[24] Ghassan Shobaki, Vahl Scott Gordon, Paul McHugh, Theodore Dubois, and Austin Kerbow. 2022. Register-Pressure-Aware Instruction Scheduling Using Ant Colony Optimization. *ACM Trans. Archit. Code Optim.* 19, 2, Article 23 (jan 2022), 23 pages. https://doi.org/10.1145/3505558

[25] Ghassan Shobaki, Austin Kerbow, and Stanislav Mekhanoshin. 2020. Optimizing Occupancy and ILP on the GPU Using a Combinatorial Approach. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3368826.3377918

[26] 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 (Feb. 2019), 30 pages. https://doi.org/10.1145/3301489

[27] Ghassan Shobaki, Laith Sakka, Najm Eldeen Abu Rmaileh, and Hasan Al-Hamash. 2015. Experimental Evaluation of Various Register-Pressure-Reduction Heuristics. *Softw. Pract. Exper.* 45, 11 (Nov. 2015), 1497–1517. https://doi.org/10.1002/spe.2297

[28] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach. *ACM Trans. Archit. Code Optim.* 10, 3, Article 14 (Sept. 2013), 31 pages. https://doi.org/10.1145/2512432

[29] Oliver Sinnen. 2014. Reducing the solution space of optimal task scheduling. *Computers & Operations Research* 43 (2014), 201–214. https://doi.org/10.1016/j.cor.2013.09.004

[30] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[31] Sid-Ahmed-Ali Touati. 2005. Register Saturation in Instruction Level Parallelism. *Intl. Journal of Parallel Prog.* 33, 4 (Aug. 2005), 393–449. https://doi.org/10.1007/s10766-005-6466-x

[32] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal Instruction Scheduling Using Integer Programming. *SIGPLAN Not.* 35, 5 (may 2000), 121–133. https://doi.org/10.1145/358438.349318