Register-Pressure-Aware Instruction Scheduling Using Ant Colony Optimization

GHASSAN SHOBAKI, VAHL SCOTT GORDON, PAUL MCHUGH, THEODORE DUBOIS, and AUSTIN KERBOW, California State University, Sacramento

This paper describes a new approach to register-pressure-aware instruction scheduling, using **Ant Colony Optimization (ACO)**. ACO is a nature-inspired optimization technique that researchers have successfully applied to NP-hard sequencing problems like the **Traveling Salesman Problem (TSP)** and its derivatives. In this work, we describe an ACO algorithm for solving the long-standing compiler optimization problem of balancing **Instruction-Level Parallelism (ILP)** and **Register Pressure (RP)** in pre-allocation instruction scheduling. Three different cost functions are studied for estimating RP during instruction scheduling. The proposed ACO algorithm is implemented in the LLVM open-source compiler, and its performance is evaluated experimentally on three different machines with three different instruction-set architectures: Intel x86, ARM, and AMD GPU. The proposed ACO algorithm is compared to an exact **Branch-and-Bound (B&B)** algorithm proposed in previous work. On x86 and ARM, both algorithms are evaluated relative to LLVM's generic scheduler, while on the AMD GPU, the algorithms are evaluated relative to AMD's production scheduler. The experimental results show that using SPECrate 2017 Floating Point, the proposed algorithm gives geometric-mean improvements of 1.13% and 1.25% in execution speed on x86 and ARM, respectively, relative to the LLVM scheduler. Using PlaidML on an AMD GPU, it gives a geometric-mean improvement of 7.14% in execution speed relative to the AMD scheduler. The proposed ACO algorithm gives approximately the same execution-time results as the B&B algorithm, with each algorithm outperforming the other on a substantial number of hard scheduling regions. ACO gives better results than B&B on many large instances that B&B times out on. Both ACO and B&B outperform the LLVM algorithm on the CPU and the AMD algorithm on the GPU.

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

Additional Key Words and Phrases: Compiler optimizations, Ant Colony Optimization, instruction scheduling, instruction scheduling for the GPU, register-pressure reduction, spill code minimization, NP-complete problems, multi-objective optimization problems

ACM Reference format:
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 (January 2022), 23 pages.
https://doi.org/10.1145/3505558

This work was supported in part by the US National Science Foundation (NSF) through Award 1911235.
Authors' address: G. Shobaki, V. S. Gordon, P. McHugh, T. Dubois, and A. Kerbow, California State University, Sacramento 6000 J Street Sacramento, CA 95819; emails: {ghassan.shobaki, gordonvs, paulmchugh}@csus.edu.

1544-3566/2022/01-ART23
https://doi.org/10.1145/3505558

1 INTRODUCTION

Register allocation and instruction scheduling are two fundamentally important compiler optimizations. In most production compilers, register allocation and instruction scheduling are done in two different passes, because doing them simultaneously in one pass would be too complex.

Instruction scheduling and register allocation are closely related, because the instruction order computed in the pre-allocation instruction scheduling pass determines the ***register pressure*** **(RP)**, which is the number of virtual registers that have overlapping live ranges and must be assigned to different physical registers. RP reflects the demand for physical registers. If the demand for registers exceeds the number of physical registers on the target machine, the register allocator must *spill* some virtual registers to main memory by adding load and store instructions (*spill code*) that may slow the program. 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 can be 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. Therefore, the impact of RP reduction on the performance of a GPU program is generally greater than its impact on the performance of a CPU program.

Minimizing RP is not the only objective of pre-allocation instruction scheduling. In fact, the original objective of instruction scheduling is exploiting **Instruction-Level Parallelism (ILP)**. ILP is exploited by executing independent instructions in parallel to minimize the schedule length, but this 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.

Compiler scheduling for ILP has a particularly high impact on the performance of in-order processors. GPUs do not reorder instructions within a single thread at run time. Therefore, the impact of compiler scheduling for ILP on GPU performance is generally higher than its impact on the performance of a modern out-of-order CPU.

The problem of balancing ILP and RP in pre-allocation instruction scheduling is a fundamental open problem in code generation and optimization. Even optimizing one of these two conflicting objectives (ILP or RP) is NP-hard [Cooper and Torczon 2011]. Current production compilers solve this problem using heuristics (usually greedy heuristics). However, recent research on both CPUs [Lozano et al. 2019, Shobaki et al. 2019] and GPUs [Rawat et al. 2018, Shobaki et al. 2020] has shown that these heuristics may produce sub-optimal results that significantly degrade performance. On GPUs, pre-allocation instruction scheduling is particularly important, because both RP and ILP significantly impact the execution time.

In the **operations research (OR)** field, researchers have successfully computed precise, and often exact, solutions to NP-hard problems using intelligent search techniques, including **Branch-and-Bound (B&B), Constraint Programming (CPR),** and **Ant Colony Optimization (ACO)**. Despite the success of these techniques in OR, applying such computationally expensive techniques to NP-hard compiler optimization problems was impractical in the past. However, today's powerful computing has motivated some researchers to explore applying some of these techniques to NP-hard problems in code optimization [Domagala et al. 2016, Lozano et al. 2018 and 2019, Shobaki et al. 2019 and 2020]. The results of this recent research show that applying such techniques can significantly improve performance in some cases and that the increase in compile time can be controlled by applying them selectively to the hot code and setting reasonable time limits [Shobaki et al. 2013 and 2019].

In this paper, we explore applying ACO to the register-pressure-aware instruction scheduling problem in compilers, and we apply it to both CPU and GPU targets. ACO is a population-based optimization technique inspired from nature. Other population-based techniques include genetic

algorithms, genetic programming, and particle swarm optimization. Ants in nature find short paths between a food source and their nest by depositing *pheromones* as they carry food. The act of ants following the pheromones while occasionally straying from the pheromone trail and the natural dissipation of pheromones cause the trail over time to shorten and approach optimality. This phenomenon inspired a class of ant-based algorithms for finding optimal solutions to NP-hard optimization problems. As detailed in Section 4, a *pheromone table* is used in an ACO algorithm to simulate the deposition and dissipation of pheromones.

ACO was introduced by Dorigo and Gambardella [1997] to compute precise solutions to large instances of the **Traveling Salesman Problem (TSP)**, which is a well-known NP-hard problem. In later research, the technique was applied to a number of related problems such as job-shop scheduling [Martens et al. 2007], protein folding [Hu et al. 2008], image processing [Jevtić 2009], and many others.

The ACO algorithm proposed in the current paper is based on the **Ant Colony System (ACS)** described by Gambardella and Dorigo [2000] for solving the ***Sequential Ordering Problem* (SOP)**, which is a generalization of the TSP. In the SOP, the objective is finding a node permutation that minimizes a path length without violating a given set of precedence constraints [Escudero 1988]. Our proposed ACO algorithm capitalizes on the similarity between the instruction scheduling problem and the SOP. The objective in both problems is finding a minimum-cost sequence that satisfies certain precedence constraints. To the best of our knowledge, our work is the first attempt to apply ACO to the register-pressure-aware instruction scheduling problem in compilers.

To apply ACO to the compiler instruction scheduling problem, the problem must be formulated as a combinatorial optimization problem with an explicit cost function. In previous work, we explored two different approaches to this two-objective optimization problem. The first approach is a single-pass approach in which the objective is minimizing a weighted sum of the schedule length and the RP cost. The second approach is a two-pass approach in which RP is treated as a primary objective that is minimized in the first pass, while schedule length is treated as a secondary objective that is minimized in the second pass. As explained in previous work [Shobaki et al. 2020], a two-pass approach is more effective on a GPU target, because optimizing occupancy is critically important and the two-pass approach ensures that enough time is spent searching for the best occupancy. In the current paper, we use the single-pass approach for CPU targets and the two-pass approach for the GPU target.

In previous work, we introduced multiple cost functions for estimating RP during instruction scheduling, including the **Peak Excess Register Pressure (PERP)** [Shobaki et al. 2013] and the **Sum of Live Interval lengths (SLIL)** [Shobaki et al. 2019] for CPU targets, and the **Adjusted Peak Register Pressure (APRP)** [Shobaki et al. 2020] for GPU targets. In this paper, we use all three cost functions. RP cost functions are summarized in Section 2.

The proposed ACO algorithm is implemented in the LLVM compiler [Lattner 2004] and its performance is evaluated on three different machines with three different instruction-set architectures: Intel x86, ARM, and AMD GPU. The proposed ACO algorithm is compared to our exact branch-and-bound algorithm [Shobaki et al. 2019, 2020]. On x86 and ARM, both algorithms are evaluated relative to LLVM's generic scheduler using the SPEC CPU 2017 benchmarks [SPEC 2017], while on the AMD GPU, the algorithms are evaluated relative to AMD's production scheduler using the PlaidML benchmarks [PlaidML]. AMD's algorithm is well-tuned for the AMD GPU.

The experimental results show that using SPECrate 2017 Floating Point (FP2017 for short), the proposed algorithm gives geometric-mean improvements of 1.13% and 1.25% in execution speed on x86 and ARM, respectively, relative to the LLVM scheduler. Using PlaidML on an AMD GPU, it gives a geometric-mean improvement of 7.14% in execution speed relative to the AMD scheduler. The ACO algorithm gives approximately the same execution-time results as the B&B algorithm,

with each algorithm outperforming the other on a substantial number of hard scheduling regions. ACO gives better results than B&B on many large instances that B&B times out on. Both the ACO algorithm and the B&B algorithm outperform the LLVM algorithm on the CPU targets and the AMD algorithm on the GPU target. An important advantage of the ACO algorithm is that it has a higher degree of parallelism and is thus more likely to benefit from parallelization on a massively parallel processor.

The rest of this paper is organized as follows. Section 2 defines the terms used in the paper and explains the preliminary concepts. Section 3 summarizes previous work. Section 4 describes the proposed algorithm. Section 5 presents the experimental results, and Section 6 summarizes the conclusions and outlines future work.

2 BACKGROUND

The problem addressed in this paper is pre-allocation instruction scheduling with RP taken into account. Scheduling is done 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 [Cooper and Torczon 2011]. The input to the instruction scheduler is a sequence of instructions with their dependencies represented by a **data dependence graph (DDG)**. The output is a schedule, which is an assignment of instructions to machine cycles. 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 used 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 with an arbitrary number of functional units and issue slots per cycle and arbitrary latencies. It also supports both pipelined and unpipelined instructions. The experimental results, however, were produced using a simple machine model. In this simple model, the processor can issue one instruction of any type in each cycle, but the model still captures instruction latencies, and this appears to be the most important factor that affects performance. In previous work, we experimented with more accurate machine models, and they did not seem to make a significant difference in terms of execution-time performance on the target processors that we experimented with.

In the pre-allocation scheduling phase, registers in the code are virtual registers. In certain special cases, the code may contain physical 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 by that instruction, and the Use set is the set of registers that the instruction uses. Our algorithm and our implementation 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 instruction that uses it has not been scheduled yet at that point.

In previous work, we used two different approaches to the pre-allocation scheduling problem: a single-pass approach [Shobaki et al. 2013, 2019] and a two-pass approach [Shobaki et al. 2020]. These approaches are described next.

2.1 Single-Pass Approach

In this approach, a weighted sum of schedule length and RP is optimized in a single pass [Shobaki et al. 2013]. Given a sequence of instructions, the objective is to find a schedule S that minimizes the following cost function:

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

where |S| is the schedule length, L_s is a lower bound on the schedule length, P is the RP cost, L_p is a lower bound on the RP cost, and w is the ***register pressure weight* (RPW)**. The RPW parameter expresses the weight of RP relative to the schedule length. A tight lower bound on the schedule length may be computed using the algorithm of Langevin and Cerny [1996]. In this work, the single-pass approach is used in scheduling for CPU targets.

2.2 Two-Pass Approach

On a GPU target, minimizing RP maximizes occupancy, and maximizing occupancy generally has a higher impact on GPU performance than exploiting ILP (minimizing the schedule length). In theory, this can be captured in the single-pass approach by setting the RPW to a sufficiently large value. Experimentally, however, we found that an extremely high RPW results in a very slow algorithm that may not spend enough time minimizing RP [Shobaki et al. 2020]. Therefore, we introduced the two-pass approach in which occupancy is maximized (RP is minimized) in the first pass as a primary objective and ILP is maximized in the second pass as a secondary objective. In the second pass, the algorithm searches for a minimum-length schedule among all the schedules that maintain the best occupancy found in the first pass. The first pass is called *the occupancy pass*, and the second pass is called *the ILP pass*. As explained in detail in the original paper, the two-pass approach is more effective for a GPU target, because it ensures that adequate time is spent optimizing occupancy. In this work, we use the two-pass approach for the GPU target.

In the two-pass approach to scheduling for the GPU, the best occupancy found in the first pass is treated as a *constraint* in the second pass. So, in the second pass, the algorithm searches for the shortest possible schedule among all the schedules that satisfy that occupancy constraint, and any schedule that does not satisfy the occupancy constraint is treated as an invalid schedule.

2.3 Register-Pressure Cost Functions

In previous work, we explored multiple cost functions for representing RP during scheduling, including the ***Peak Excess Register Pressure (PERP)*** [Shobaki et al. 2013] and the ***Sum of Live Interval Lengths (SLIL)*** [Shobaki et al. 2019] for CPU targets and the ***Adjusted Peak Register Pressure* (APRP)** [Shobaki et al. 2020] for GPU targets. In this subsection, we briefly describe these cost functions. The details can be found in the original papers.

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 available physical registers of that data type on the target machine.

Assuming that the code is in **Static Single Assignment (SSA)** form [Cooper and Torczon 2011], each virtual register in a given 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.

As explained in previous work [Shobaki et al. 2019], the SLIL cost function may capture live interval overlaps that are not captured by the PERP cost function. SLIL captures the overlaps among all intervals, while PERP captures only the overlaps that contribute to the peak pressure. In a high-pressure scheduling region, the peak-pressure point in the schedule is not the only point that will cause the register allocator to insert spill code. Therefore, minimizing SLIL is more likely to minimize spill code than minimizing PERP, especially in larger scheduling regions with multiple high-pressure segments.

On a GPU, multiple PRP values may give the same occupancy value. To account for this, we introduced the **adjusted peak register pressure (APRP)** step function for modeling occupancy during instruction scheduling. The APRP of a given PRP value x is the maximum PRP value 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 RELATED WORK

Compiler researchers have been studying instruction scheduling for many decades. Instruction scheduling for minimum register pressure, or the **Minimum-Register Instruction Sequence (MRIS)** problem, has been studied since 1970 by Sethi and Ullman [1970] who proposed an algorithm for finding an instruction order for computing an expression using the minimum number of registers when the DDG is a tree. However, a tree DDG is a special case with limited practical value, as the DDGs constructed in an optimizing compiler for real code are often not trees.

A more practical algorithm for balancing ILP and RP was proposed by Goodman and Hsu [1988]. That algorithm was heuristic-based with no guarantee of optimality. Other heuristic approaches were then proposed by Govindarajan et al. [2003], Touati [2005] and Barany and Krall [2013].

Over the past two decades, some researchers proposed combinatorial approaches (intelligent search techniques) that are guaranteed to give optimal solutions if they terminate within a given time limit. Kessler [1998] proposed a dynamic programming approach to both the MRIS problem and the general problem of balancing RP and ILP. Barany and Krall [2013] proposed an Integer Linear Programming solution to the MRIS problem. Malik [2008] used **Constraint Programming (CPR)** to solve the MRIS problem, and Domagala et al. [2016] used CPR to integrate RP-aware instruction scheduling and loop unrolling. Lozano et al. [2018] used CPR to solve the integrated instruction scheduling and register allocation problem. Lozano provides an excellent survey of combinatorial approaches to instruction scheduling and register allocation [Lozano 2019].

Shobaki et al. [2013, 2019, 2020] presented a B&B algorithm for solving the RP-aware instruction scheduling problem. In the experimental evaluation, we compare the proposed ACO algorithm with that B&B algorithm. B&B is an exact technique for solving combinatorial optimization problems by conducting an exhaustive search of the solution space. The exhaustive search may complete in reasonable time on many instances if powerful pruning techniques are used to prune non-promising sub-spaces as early as possible. Shobaki et al. successfully applied this technique to RP-aware instruction scheduling and showed that it can in some cases produce a significant performance improvement relative to the greedy heuristics used in production compilers. However, that B&B algorithm times out on thousands of instances in SEPC CPU2006.

Rawat et al. [2018] describe a source-level re-ordering algorithm to minimize RP for stencil computation on the GPU. Using a pattern-specific approach, they report speedups in the range of 1.22x to 2.43x for NVCC and 1.15x to 2.08x for LLVM. These results show the significant impact of RP-aware scheduling on GPU performance.

ACO was introduced in 1992 by Marco Dorigo in his doctoral thesis [Dorigo 1992] and later published (as Ant Colony System) in the first issue of IEEE Transactions on Evolutionary Computation [Dorigo and Gambardella 1997]. The technique was originally applied to the TSP. Our proposed algorithm is based on the ACO proposed by Gambardella and Dorigo for the SOP [2000]. Using an ACO approach to solve other scheduling problems was studied by Ferrandi et al. [2010] and Wang et al. [2007]. Both explored the use of ACO for operation and resource scheduling, a somewhat more general problem of scheduling resources in software systems. They both noted

the applicability to compilers but did not actually implement their algorithms in compilers. Those works, however, focused on minimizing the schedule length (exploiting ILP) without considering RP. To our knowledge, our proposed algorithm is the first ACO algorithm for RP-aware instruction scheduling.

More direct applications of ACO to compiler optimization have been proposed by Lintzmayer et al. [2012] and de Souza Xavier et al. [2018]. Lintzmayer et al. used an ACO approach to do graph coloring register allocation, and de Souza Xavier et al. used ACO to do design space exploration. Both reported good results.

The pre-allocation scheduling problem addressed in this paper is a **multi-objective optimization problem (MOOP)**. Researchers have used various approaches to MOOPs. A common approach is to compute a weighted sum of the different objectives, thus reducing the problem from a multi-objective problem to a single-objective problem. This approach has been used in a number of ACO applications [Alaya 2007, Infante 2010]. Another approach that has been used in ACO applications is the Tchebycheff *weighted metric* approach that is based on identifying an optimum for each objective, forming an idealized (utopian) solution and then finding the nearest feasible solution to the idealized solution [Mu 2019]. A third approach to MOOPs is searching for *Pareto optimal* solutions. A solution to a MOOP is pareto optimal if no objective can be improved without degrading at least one other objective.

Researchers have used multiple techniques to adapt ACO to solving MOOPs. Whereas some researchers used a single pheromone table as in a standard ACO algorithm, other researchers used multiple pheromone tables, usually one per objective [Gambardella 1999, Doerner 2004, Infante 2010]. Some algorithms keep track of the Pareto optimal solutions found [Liu 2019, Mu 2019], and others limit pheromone table updates to favor those solutions that are Pareto optimal [Alaya 2007]. Reviews of various ACO approaches to solving MOOPs can be found in the papers of Leguizamón [2010] and Ning [2019].

4 ALGORITHM DESCRIPTION

ACO is inspired by the way ants utilize pheromones to construct trails. In ACO, artificial ants traverse paths in a graph in a probabilistic manner. The quality of a generated path determines the amount of pheromone deposited on each of the arcs that constitute that path. Subsequent ants then select the sequence of nodes they visit based on the amount of pheromone on each possible arc as well as a problem-specific heuristic, which we call the ***guiding heuristic***. It is also customary to simulate the natural gradual dissipation of pheromone.

4.1 ACO Applied to Instruction Scheduling

Our proposed algorithm is based on the ACO algorithm proposed by Gambardella and Dorigo [2000] for the SOP. This particular version of ACO was chosen because of the similarities between instruction scheduling and the SOP. In both problems, the input is a DDG representing precedence constraints, and the objective is finding an order that minimizes a certain cost function. However, the RP-aware instruction scheduling problem is more complex than the SOP, because it is a MOOP. In our single-pass approach, using a weighted sum of two objectives (the RP cost and the schedule length) adds more complexity to the problem, while in our two-pass approach, the additional complexity arises from treating RP as a constraint in the second pass. Another complexity in both approaches is the presence of latency constraints.

The algorithm generates a large number of candidate schedules. Each candidate schedule is a complete sequence of instructions for the given scheduling region. Whenever a better candidate schedule is found, it is saved as the best schedule found so far. The algorithm terminates when a certain number of iterations N have been performed without finding an improvement of the best

schedule. This number of iterations is called the ***termination condition***. The termination condition is used to control the amount of time given to the ACO algorithm to find a good solution. In the context of compiler instruction scheduling, the termination condition is used to control compile time. In the experimental evaluation, we show how the termination condition affects performance.

ACO uses a pheromone table to guide the construction of candidate schedules. The pheromone table is a two-dimensional table with *n* rows and *n* columns, where *n* is the number of instructions in the current scheduling region. For any two instructions *i* and *j*, the value τ_{ij} in the pheromone table is the amount of pheromone placed on the arc between instructions *i* and *j*. If *j* is a root node in the DDG, a pseudo-instruction i_0 is used in the pheromone table instead of *i*.

A candidate schedule is initially empty. It is built by selecting one instruction at a time until the schedule is complete. At each step during schedule construction, the next instruction is selected from the ***ready list***, which is a list of the unscheduled instructions that have had all of their dependencies satisfied. The ready list is updated each time an instruction is added to the schedule, as scheduling an instruction may make some of its dependents ready.

The choice of the next instruction to select (i.e., the next link that an ant traverses) is done randomly, but with a bias that takes into account both the values in the pheromone table and the guiding heuristic. For each instruction i, η_i is the value of the guiding heuristic for i. η_i is encoded as a number in the range [1, 2] with larger values representing higher priorities. Our implementation supports multiple guiding heuristics as detailed in Section 4.2. Each instruction *i* in the ready list is assigned a score $\tau_{i=} \tau_{li}\eta_i$, where *l* is the previous instruction selected. Then the next instruction is selected using one of two methods:

- —First method with probability *s/n*: The next instruction is selected through biased random selection, where the probability of selecting an instruction from the ready list is proportional to its τ_i. In population-based optimization algorithms, this approach is commonly referred to as ***fitness-proportional selection***.
- —Second method with probability 1-*s/n*: The next instruction is the instruction with the maximum τ_i.

In the above probabilities, *s* is an adjustable parameter used to control the balance between ***exploitation*** and ***exploration***. The value *s* represents the average number of instructions selected through biased random selection (exploration) as opposed to strict pheromone-based selection (exploitation). For example, if *n* is 100 and *s* is 20, 20% of the selections will be made based on biased randomness (exploration) and 80% of the selections will be made based on the highest pheromone value (exploitation). In population-based optimization systems, exploration refers to using individuals in the population (in this case artificial ants) to examine previously untested possibilities, while exploitation refers to using individuals in the population to continue examining possible solutions that are similar to the better solutions that have been discovered so far. Experimentally, s = 10 for any *n* gave better results than any other setting that we tried.

Each iteration simulates a certain number of ants, and each ant generates a candidate schedule. As shown in the experimental evaluation, we experimented with multiple settings of the number of ants per iteration. At the end of each iteration, the pheromone table is updated based on the best schedule in that iteration (the iteration winner). In this update, the pheromone on each link (i, j) in the iteration's best schedule is incremented according to the following formula:

$$\tau_{ij} += \max\left(\left(1 - \frac{C_{best}}{k * C_{heur}}\right)\ (d_{max} - d_{min}),\ 0\right) + d_{min} \tag{2}$$

where C_{best} is the cost of the iteration's best schedule, C_{heur} is the cost of the initial heuristic schedule, d_{min} and d_{max} are the minimum and maximum amounts of pheromone that can be deposited,

and k is a tuning parameter. Experimentally, the best results were obtained with k = 1.5, $d_{min} = 1$, and $d_{max} = 6$.

It is important to note here that in the second pass of the two-pass version of our algorithm, the iteration's best schedule is selected from the schedules that satisfy the occupancy constraint. If no schedule in a given iteration satisfies the occupancy constraint, no pheromone table update will take place at the end of that iteration.

In order to simulate the decay of pheromones over time, the following formula is applied to each link in the pheromone table at the end of each iteration:

$$\tau_{ij} = \min(\max(\tau_{ij}(1-\rho), \tau_{min}), \tau_{max}) \quad (3)$$

where ρ is the decay rate, τ_{min} and τ_{max} are the minimum and maximum amounts of pheromone that a link in the table can have. Experimentally, the best results were obtained with $\rho = 0.1$, $\tau_{min} = 1$, $\tau_{max} = 8$.

4.2 Heuristics

In the proposed ACO algorithm, various heuristics are used to both construct the initial solution and guide the ACO search, including the **Critical-Path (CP)** heuristic [Cooper and Torczon 2011] and the **Last Use Count (LUC)** heuristic described in previous work [Shobaki et al. 2015].

The CP heuristic is a commonly used heuristic for minimizing schedule length (exploiting ILP). The CP of an instruction is the length of the longest path between the instruction and a leaf node in the DDG. Thus, the CP measures the length of the dependence chain below an instruction. When multiple instructions are ready, selecting the instruction with the longest dependence chain below it increases the chances of hiding long latencies and consequently minimizing the schedule length.

The LUC heuristic is an intuitive heuristic for minimizing register pressure. The LUC of an instruction is the number of live ranges that the instruction closes. When multiple instructions are ready, selecting the instruction that closes the maximum number of live ranges is likely to minimize register pressure.

Both the CP and LUC are greedy heuristics that are not guaranteed to produce optimal solutions, even if we consider a single objective (schedule length or RP). Of course, the problem of optimizing two conflicting objectives is much more complex than the problem of minimizing one objective.

LLVM's scheduling algorithm is a list scheduling algorithm [Cooper and Torczon 2011] that is based on maintaining a ready list of instructions and selecting the next instruction to add to the schedule according to certain priority schemes, including schemes that are similar to LUC and CP. The LLVM algorithm gives higher priority to reducing register pressure, and thus its behavior tends to be similar to that of the LUC heuristic, that is, it produces low-RP schedules that can be too long. The AMD algorithm is based on the LLVM algorithm. It extends the LLVM algorithm to produce better schedules for an AMD GPU by balancing RP and ILP. It also involves multiple AMD-GPU-specific enhancements.

5 EXPERIMENTAL RESULTS

The proposed ACO algorithm was implemented in the LLVM compiler as an alternative pre-allocation scheduler. In this section, we present the results of our experimental evaluation. The proposed ACO algorithm was compared to the exact B&B algorithm proposed in previous work [Shobaki et al. 2013, 2019]. To show the importance of RP awareness, the evaluation also includes the CP algorithm that only considers ILP and does not consider RP.

Table 1. Machine Configurations and Benchmarks

Architecture	Processor	Benchmarks	LLVM Version	OS
X86	Intel(R) Core (TM) i9-9900X @ 3.50 GHz	FP2017	LLVM 7.0	Ubuntu 18.04.5
ARM	Broadcom BCM2711 Cortex A-72 @600 MHz (underclocked) Cross compilation was done on an Intel Core i7-7700K processor @ 4.2 GHz	FP2017	LLVM 7.0	Ubuntu 20.04.1 LTS
AMD GPU	AMD Radeon RX Vega 64 GPU @ 1.63 GHz Compilation was done on an AMD Ryzen Threadripper 1950X processor	PlaidML	roc-ocl-2.4.0	Ubuntu 18.04.5

5.1 Experimental Setup

The ACO scheduler and the B&B scheduler were implemented in LLVM as alternative schedulers to LLVM's machine-level pre-allocation scheduler. Three target architectures are included in the evaluation: x86-64, ARM, and an AMD GPU. For x86 and ARM, both algorithms are evaluated relative to LLVM's generic scheduler, while for the AMD GPU, the algorithms are evaluated relative to AMD's production scheduling algorithm, which is an extension of the LLVM algorithm. For the CPU targets, the single-pass version of the algorithm was used, while for the GPU target, the two-pass version was used.

The hardware and software configurations and the benchmarks used for each target architecture are shown in Table 1. The benchmarks used in the evaluation are SPECrate 2017 Floating Point[1] (FP2017) for CPU targets and PlaidML for the GPU target. The clock speed for the ARM target was reduced to 600 MHz to avoid the thermal throttling that caused random variation in execution times. The −O3 optimization level was used in all tests. At this optimization level, LLVM invokes a global greedy register allocator.

As shown in the next subsections, we experimented with different settings for the termination condition and the number of ants per iteration in the ACO algorithm. However, the base settings were as follows. On CPU targets, the termination condition was set to 50 iterations, and the number of ants was set to 10 ants per iteration. On the GPU target, the termination condition was set to 10 in the occupancy pass and to 5 in the ILP pass, and the number of ants was set to 10 in the occupancy pass and to 40 in the ILP pass. For the B&B algorithm, the time limit was set to 10ms/instruction for the CPU targets and 1ms/instruction in each pass for the GPU target.

For CPU targets, the LUC heuristic was used both as an initial heuristic and as a guiding heuristic. For the GPU target, LUC was used as an initial heuristic and as a guiding heuristic in the occupancy pass, and CP was used as the guiding heuristic in the ILP pass.

5.2 Benchmark Statistics

Table 2 shows some statistics about the benchmarks used in our experimental evaluation. In FP 2017, there are 12 benchmarks that have tens of thousands of functions. Each function is divided into scheduling regions. In most cases, a scheduling region is a basic block, but in some cases, LLVM may divide a basic block into multiple scheduling regions based on target-specific considerations. The total number of scheduling regions is 1,123,793. Each scheduling region is an instance of the scheduling problem.

The average instance size in FP2017 is 7.1 instructions, which indicates that there are many small instances that can be easily scheduled optimally. However, the last row in the table shows

[1]The wrf benchmark was excluded due to a run-time error that was seen even with the base LLVM compiler.

Table 2. Benchmark Statistics

STAT	FP2017	PlaidML
Number of benchmarks	12	13
Number of functions/kernels	66,422	3,814
Number of scheduling regions	1,123,793	16,682
Avg. scheduling regions per function	16.9	4.4
Avg. Instructions per scheduling region	7.1	49
Max. Instructions per scheduling region	6193	921

Table 3. Reductions in the RP Cost Functions

ALGORITHM	PERP	SLIL	APRP
B&B	2.38%	43.00%	17.94%
ACO	1.11%	39.93%	18.98%
CP	−18.96%	−140.76%	−23.15%

that there are large instances with up to 6,193 instructions, which is quite large for an NP-hard problem.

In PlaidML, there are 13 benchmarks that have 3,814 kernels and 16,682 scheduling regions. The average region size is 49 instructions, which is significantly larger than the average region size in FP2017. This is attributed to the fact the GPU applications have, on average, more straight-line code and fewer conditionals.

5.3 Register-Pressure Cost

The experiments in this section focus on register-pressure reduction. On the CPU targets, ILP is ignored by setting all latencies to one. On the GPU target, latencies are naturally ignored in the occupancy pass.

Table 3 shows the percentage reductions in the different RP cost functions produced by each algorithm under study relative to the base algorithm. For PERP and SLIL, the base algorithm is LLVM's scheduling algorithm, and for APRP, the base algorithm is AMD's scheduling algorithm. The PERP and SLIL results are for the Intel x86 target, while the APRP results are for the AMD GPU target. The numbers in the table are the percentage reductions in the overall RP cost across all the scheduling regions in each benchmark suite. Recall that PERP and SLIL are used to compile FP2017 for CPU targets and APRP is used to compile PlaidML for the GPU target.

The first observation about the results in Table 3 is that the reductions in the SLIL cost function are greater in magnitude than the reductions in the PERP cost function. This is attributed to the fact that the SLIL cost function captures any reduction in live interval overlapping, whether the RP is above the physical limit (the number of physical registers in the target machine) or not. On the other hand, PERP accounts only for RP reductions above the physical limit. Therefore, all the reductions in PERP are expected to cause reductions in spill code, while not all the reductions in SLIL are expected to cause reductions in spill code. The reductions in APRP are greater than the reductions in the PERP, because the average scheduling region size in PlaidML is greater than the average region size in FP2017 (see Table 2).

The numbers in Table 3 show that ACO and B&B generally give comparable reductions in all RP cost functions. B&B gives greater reductions in PERP and SLIL, but ACO gives a greater reduction in APRP. As expected, the CP algorithm, which maximizes ILP without considering RP, significantly increases all RP cost functions relative to the base algorithms. CP is included in our

Table 4(a). Instance-level ACO vs B&B comparison in FP2017 using SLIL on Intel x86

DESCRIPTION	COUNT	Percentage
1. Total instances passed to B&B and ACO	561,131	
2. Optimal by both (easy)	511,491	91.2% of total
3. Hard instances	49,640	8.8% of total
4. Optimal by B&B only	37,738	76% of hard
5. B&B timeouts	11,902	24% of hard
6. B&B timeouts with ACO and B&B equal	815	7% of timeouts
7. B&B timeouts with B&B better	6,552	55% of timeouts
8. B&B timeouts with ACO better	4,535	38% of timeouts

Table 4(b). Difference in SLIL Cost Relative to B&B

ALGORITHM	Difference in SLIL cost relative to optimal B&B
ACO	6.44%
LLVM	175.2%
CP	396.2%

experimental evaluation for sanity checking. It shows that the base algorithms have good performance and that both ACO and B&B perform significantly better than good base algorithms.

Table 4(a) shows a comparison between ACO and B&B at the instance level in FP2017 using the SLIL cost function on Intel x86. Each instance is a scheduling region in FP2017. After filtering out the trivial instances that were solved optimally by the heuristic, 561,131 instances were passed to the search algorithm (B&B or ACO). Row 2 shows that 91.2% of these instances were solved optimally by both algorithms (easy instances). The remaining 8.8% of the instances are considered *hard* instances. Although the hard instances constitute only 8.8% of the total number of instances, they are likely to have a higher impact on performance than the easy instances, because hard instances tend to be larger scheduling regions with more substantial code. Given a set of scheduling regions with the same execution frequency, larger scheduling regions are likely to have a higher impact on a program's performance.

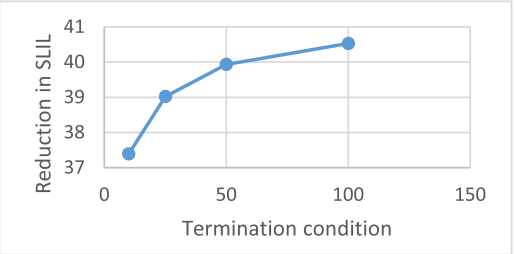
Row 4 in Table 4(a) shows that 76% of the hard instances were solved optimally by B&B but not by ACO (ACO gave larger cost values). The last three rows show a comparison between ACO and B&B on the 11,902 instances on which B&B times out (the hardest instances). Row 6 shows that both algorithms produced the same results for 7% of these timeouts. Row 7 shows that B&B produced better results for 55% of the timeouts, and Row 8 shows that ACO produced better results for 38% of the timeouts. Instances that time out tend to be larger instances. This result shows that ACO has an advantage over B&B on thousands of hard instances that the B&B algorithm could not solve optimally within the time limit.

Table 4(b) shows the percentage difference in SLIL cost between each of ACO, LLVM and CP relative to B&B on the 549,229 instances that were solved optimally by B&B. The results in the table show that the costs produced by ACO are much closer to optimal than the costs produced by the other two scheduling algorithms. It should be noted here that differences in SLIL do not necessarily lead to differences in spilling, as different schedules with a peak RP below the physical limit may have different SLIL costs. To better understand this, the interested reader is referred to the paper that introduces the SLIL cost function [Shobaki et al. 2019].

Table 5 shows the impact of the termination condition on the performance of the ACO algorithm on Intel x86 using FP2017. The table shows the reduction in SLIL relative to LLVM for ACO

Table 5. Impact of the Termination Condition

TERMINATION CONDITION	REDUCTION IN SLIL
10	37.39%
25	39.02%
50	39.93%
100	40.53%



with termination conditions of 10, 25, 50, and 100 iterations without an improvement. Ten ants per iteration were used. The results in this table show that increasing the termination condition value results in measurable reductions in SLIL, but even with a termination condition of 10, the ACO algorithm produces a significant reduction in SLIL relative to LLVM. In practice, a larger termination condition value increases compile time. Therefore, the objective is achieving the best execution-time results using the minimum termination condition value.

5.4 Spills Generated by the Register Allocator

In this subsection we study the impact of the RP reductions reported in the previous subsection on the amount of spilling. The target processor is Intel x86 and the benchmarks are FP2017. The SLIL cost function is used in this experiment, as previous work showed that SLIL gives a stronger correlation with the amount of spill code [Shobaki et al. 2019]. Similar to the previous subsection, ILP is ignored.

Unlike the instruction scheduler, which operates on one scheduling region at a time, the register allocator at higher optimization levels in many production compilers, including LLVM, operates on a whole function (global register allocation). This further complicates the already complex relation between the RP of the schedules and the number of spills generated by the register allocator. For example, a global register allocator is designed to avoid generating spills in basic blocks with higher execution frequencies (inside loops) and favor spilling in basic blocks with lower execution frequencies (outside loops). The instruction scheduler is not aware of execution frequencies. Consequently, the instruction scheduler may minimize RP within a low-frequency block, but the register allocator may still generate significant spill code in that block to avoid spilling in blocks with higher frequencies.

Due to the complexity of the relation between RP and the number of spills, the impact of the scheduler's RP reduction must be studied using a fairly large dataset to establish statistical significance. In this paper, we use a metric that was introduced in previous work [Shobaki et al. 2015] for experimentally evaluating the effectiveness of a scheduling algorithm at minimizing spilling. This metric is an experimental lower bound on the size of the gap between the performance of the algorithm under study and optimal performance. 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:

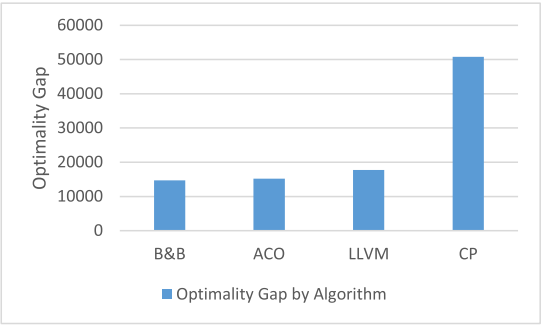
$$SOM\,(F, A) = \sum_{F} \min_{A} (spills\,(f, a)) \tag{4}$$

Table 6. Spill-Code Statistics for FP2017

STAT	VALUE
Total functions	66,422
Functions with spills	12,096 (18.2%)
Sum of minima (SOM)	256,738
Avg. spills per function	3.9
Avg. spills per spilling function	21.2

Table 7. Optimality Gaps

Algorithm	Optimality Gap	%Funcs at Min	Max Extra Spills
B&B	14,677 (5.72%)	80.70%	239
ACO	15,196 (5.92%)	78.87%	368
LLVM	17,733 (6.91%)	76.36%	355
CP	50,800 (19.79%)	72.60%	4,444



where *spills(f, a)* is the number of spills generated by the register allocator when algorithm $a \epsilon A$ is used to schedule the regions in function $f \epsilon F$. As explained in previous work [Shobaki et al. 2015], the *SOM* is an upper bound on the optimal sum of spills across 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)*. For a given algorithm *a* in a set of algorithms *A*, the difference between *a*'s spill sum and the SOM is

$$ExtraSpills\,(a, A) = \sum_{F} spills\,(f, a) - SOM\,(F, A) \tag{5}$$

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*.

Each algorithm under study was applied to all the functions in FP2017, and the number of spills generated by LLVM's register allocator at −O3 was used to evaluate the scheduling algorithms' effectiveness at reducing RP. Table 6 shows the spill-code statistics in FP2017. The third row shows that 18% of the functions in FP2017 have spills, and the fourth row shows that the SOM is 256,738 spills. This SOM was computed using 13 algorithms, including the proposed ACO algorithm, the B&B algorithm with different parameter values as well as a collection of heuristics.

Table 7 shows the spill statistics for the algorithms under study. The second column in the table shows the optimality gap, which is the number of extra spills produced by each algorithm relative to the SOM. The third column shows the percentage of functions in which each algorithm produced the same number of spills as the minimum number produced by any algorithm in that function. The last column shows the maximum number of extra spills relative to the SOM that each algorithm produced in any function (the worst-case). The extra spills for B&B and ACO are approximately the same. Both B&B and ACO produced thousands of spills less than those produced by the LLVM scheduler. As expected, the CP algorithm, which does not take RP into account, produced an excessive amount of spilling. In the worst case, CP produced 4,444 extra spills in

Table 8. Comparison between ACO and B&B on Occupancy in PlaidML

Algorithm	B&B	ACO
Average occupancy	7.79	7.82
%Improvement relative to AMD	6.6%	7.0%
Kernels with better occupancy	218 (5.7%)	339 (8.9%)

one function. Comparing the results in Table 7 with those in Table 3 shows a strong correlation between the SLIL cost and the amount of spilling.

Although the correlation between the scheduling cost and the amount of spilling is strong, it is not perfect. For example, ACO's optimality gap is significantly smaller than LLVM's, but LLVM's worst-case behavior on any instance (last column) is slightly better than ACO's worst-case behavior. This less-than-perfect correlation is attributed to the complexity of the relation between instruction scheduling and register allocation. In a production compiler, instruction scheduling and register allocation are two different NP-hard problems that are solved separately (due to the complexity of solving them simultaneously), and each problem is solved in a different scope. The scope of the instruction scheduler is a scheduling region (usually a basic block), while the scope of the register allocator is a whole function. Since the problem is NP-hard, the register allocator may produce sub-optimal solutions for many instances. Due to this sub-optimality, the register allocator may generate too many spills in a code section in which the scheduler has actually minimized RP.

We note here that although the reduction in spilling produced by the proposed ACO algorithm relative to LLVM is significant, it is still limited. This is due to the fact that the LLVM scheduler for Intel x86 is heavily biased towards minimizing RP. The execution-time results in Section 5.6 show that the proposed ACO algorithm achieves a better balance between RP and ILP, and thus produces better execution-time results than the LLVM algorithm.

5.5 GPU Occupancy and Schedule Length

On the GPU target, the main purpose of reducing RP is increasing occupancy rather than reducing spilling. Furthermore, reducing schedule length is important for a GPU target, because a GPU does not have out-of-order execution within a thread. In this section, we study the impact of the proposed ACO algorithm and the B&B algorithm on occupancy and schedule length.

Table 8 shows a comparison between the occupancies produced by ACO and B&B on the PlaidML benchmarks. On average, ACO gives a slightly better occupancy than B&B (7.82 compared to 7.79). Both algorithms give significantly higher average occupancies than the AMD algorithm (ACO is 7.0% better and B&B is 6.6% better). ACO gives a better occupancy than B&B on 339 kernels (8.9% of the kernels), while B&B gives a better occupancy than ACO on 218 kernels (5.7% of the kernels). On the rest of the kernels, both algorithms give the same occupancy. These results show that ACO performs significantly better than B&B in the occupancy pass.

Before we compare the performance of the proposed ACO algorithm with that of the B&B algorithm in the ILP pass, we study the effect of the termination condition and the number of ants on the schedule length produced by the ACO algorithm in that pass. Table 9 shows the average schedule length across all the scheduling regions in PlaidML using different termination conditions and different numbers of ants per iteration. Recall that the base setting used in this paper for the ILP pass is 40 ants per iteration and a termination condition of 5 iterations without improvement. The base setting is shown in boldface in the table.

The results in Table 9 show that increasing the termination condition and the number of ants beyond the base setting produces a significantly better average schedule length. With a termination condition of 50 iterations and 800 ants per iteration, an average schedule length of 151.89 is

Table 9. Schedule Length Dependence on the Number of Ants and the Termination Condition

Termination Condition	Number of Ants	Avg. Sched. Length
5	10	178.73
5	**40**	**165.16**
5	50	164.12
5	100	160.91
5	200	158.75
50	10	163.10
50	50	159.49
50	100	157.66
50	200	155.69
50	800	151.89

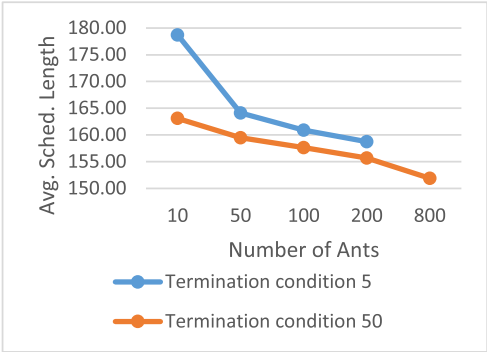


Table 10(a). Comparison between Base ACO and B&B on Schedule Length in PlaidML

Algorithm	B&B	ACO
Average schedule length	150.4	165.2
%Improvement relative to AMD	6.65%	−2.88%
Regions with better schedule length (won)	4,200	318
Avg. size of regions won	116.8	230.4
Avg. win margin	74.4	208.4

Table 10(b). Comparison between Many-Ant ACO and B&B on Schedule Length in PlaidML

Algorithm	B&B	ACO many ants
Average schedule length	150.4	151.9
%Improvement relative to AMD	6.65%	5.60%
Regions with better schedule length (won)	2,381	350
Avg. size of regions won	164.1	232.4
Avg. win margin	63.1	357.9

achieved. As shown in Section 5.7 this setting results in an extremely large compile time. However, this result is still interesting, because it indicates that much better performance of the ACO algorithm can possibly be achieved in the future by developing a parallel version of the algorithm that runs on a massively parallel processor. The ACO algorithm is naturally parallelizable, because each ant independently constructs a different schedule. This suggests that it should be possible to use a massively parallel processor to run many ants in parallel and produce a significantly better schedule in much less compile time than that of the current sequential algorithm. Parallelizing the ACO algorithm is beyond the scope of the current paper.

Table 10 compares the performance of ACO with B&B in the ILP pass. Table 10(a) shows the comparison for the ACO algorithm with the base setting (termination condition of 5 and 40 ants), while Table 10(b) shows the comparison for the many-ant setting (termination condition of 50 and 800 ants). Comparing the average schedule lengths in Row 2 shows that the B&B algorithm gives a substantially better schedule length than the ACO algorithm with the base setting. This is attributed to the fact that the search in the ILP pass is constrained by maintaining the best APRP

found in the occupancy pass. This constraint allows the B&B algorithm to search the solution space faster, because B&B has the capability to backtrack as soon as the APRP exceeds the target APRP. On the other hand, the ACO algorithm does not backtrack and can only terminate the current ant as soon as the APRP exceeds the target.

Table 10(b) shows that increasing the number of ants narrows the gap between the ACO algorithm and the B&B algorithm. With 800 ants and a termination condition of 50 iterations, the ACO algorithm produces approximately the same average schedule length as the B&B algorithm.

The fourth row in each table shows the number of scheduling regions that each algorithm *won*, that is, the number of regions for which each algorithm produced a better schedule. The last two rows show the average size of the regions won by each algorithm and the average win margin. Although B&B wins more regions than ACO, the regions won by ACO are, on average, much larger, and ACO wins by larger margins, which is a great advantage of the ACO algorithm relative to the B&B algorithm. This is attributed to the fact that the B&B algorithm systematically explores the solution space, while the ACO algorithm has a degree of randomness that allows it to explore non-adjacent solutions. On larger solution spaces, the B&B is more likely to get trapped in a small sub-space, while the randomness of ACO allows it to explore solutions in many different subspaces. This result suggests that a hybrid algorithm that combines the advantages of B&B and ACO is likely to give better results than either algorithm can give individually. We plan to explore such a hybrid algorithm in future work.

5.6 Execution Times

The statistics reported in the previous subsections show that the proposed ACO algorithm improves important compile-time metrics that affect a program's performance, including the amount of spilling, GPU occupancy, and schedule length. In this section, we study the impact of these factors on the actual execution times of CPU and GPU programs.

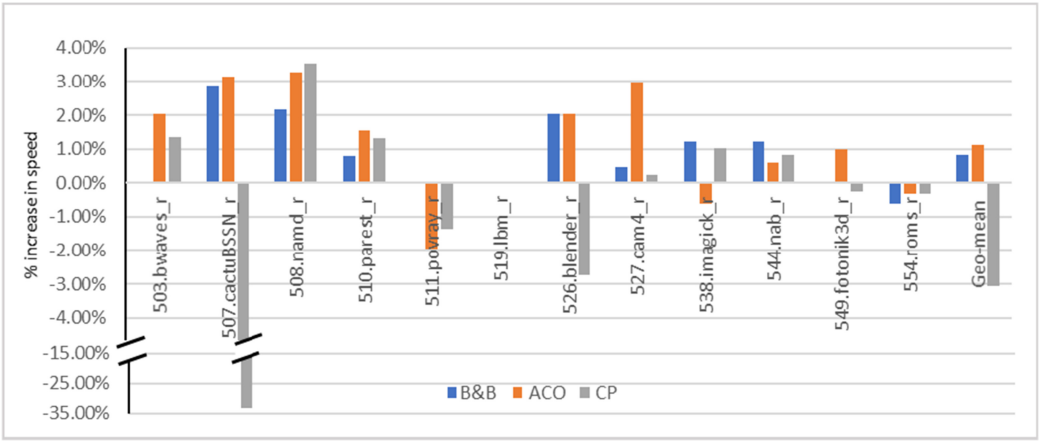
It is important to note here that the relation between the compile-time metrics reported in the previous subsections and the execution time is fairly complicated. The scheduler's objective is balancing ILP and RP, but these are only two out of many factors that affect the execution time. The execution time depends on a complex combination of many different factors, some of which cannot be modeled accurately during compilation. Examples of such factors include memory bandwidth, caching performance, and structural hazards in the processor's pipeline. In practice, each compiler optimization pass is designed to improve one or two factors that affect performance and uses certain heuristics to minimize the negative side effects on other factors. Clearly, we can't expect such heuristics to totally eliminate negative side effects, because some of the factors that affect performance are hard to model accurately and the interactions among the different optimizations are hard to predict. Therefore, the negative side effects of a compiler optimization are often unavoidable in practice, and the execution times reported in the current section cannot be fully explained based only on the two factors that are controlled by the scheduling algorithm.

To achieve execution-time improvements with minimum increase in compile time for the CPU targets, the ACO algorithm and the B&B algorithm were applied only to the ***hot functions*** in each benchmark. A hot function is a function in which the program spends a significant percentage of its execution time. In this experimental evaluation, the *Perf* tool [Perf] was used to profile the benchmarks and identify hot functions. Every function in which a benchmark spends 2% or more of its execution time was considered hot. For the GPU target, the ACO and the B&B algorithms were applied to all the GPU kernels in each benchmark.

For the CPU targets, SPEC rate tests were run using eight threads on Intel x86 and one thread on ARM. One thread was used on ARM, because the machine has limited memory. The register pressure weight in Equation (1) was set to 100 for both targets.

Table 11. FP2017 on Intel Percentage Increase in Execution Speed Relative to LLVM's Default Scheduler

BENCHMARK	B&B	ACO	CP
503.bwaves_r	0.00%	2.03%	1.35%
507.cactuBSSN_r	2.85%	3.13%	-33.33%
508.namd_r	2.17%	3.26%	3.53%
510.parest_r	0.78%	1.56%	1.30%
511.povray_r	0.00%	-1.96%	-1.38%
519.lbm_r	0.00%	0.00%	0.00%
526.blender_r	2.05%	2.05%	-2.74%
527.cam4_r	0.46%	2.97%	0.23%
538.imagick_r	1.21%	-0.60%	1.01%
544.nab_r	1.22%	0.61%	0.81%
549.fotonik3d_r	0.00%	1.00%	-0.25%
554.roms_r	-0.61%	-0.31%	-0.31%
Geo-mean	**0.84%**	**1.13%**	**-3.05%**



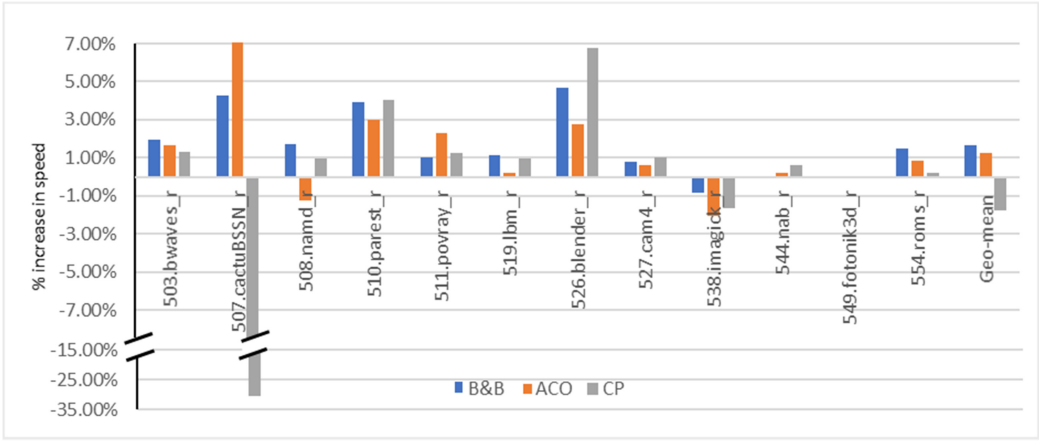
Tables 11, 12, and 13 show the percentage gains in execution speed produced by the proposed ACO algorithm, the B&B algorithm, and the CP algorithm relative to the base algorithm for the Intel, ARM, and AMD GPU targets, respectively. For FP2017, the execution speed is measured by the SPECrate score, while for PlaidML, the execution speed is measured by tiles per second. For the CPU targets, the gains are relative to the LLVM scheduling algorithm, while for the GPU target, the gains are relative to AMD's production scheduling algorithm. Positive numbers in the tables indicate performance gains, and negative numbers indicate regressions. Usually, there is random variation in execution time. The numbers in the tables are based on the median of three runs. The last row in each table shows the geometric-mean improvement for each algorithm across the *entire* benchmark suite.

Overall, the proposed ACO algorithm and the B&B algorithm give significant performance gains relative to the base algorithms on all three targets. The proposed ACO algorithm gives a slightly greater performance gain on Intel and the AMD GPU and a slightly smaller gain on ARM. The CP algorithm that does not consider RP gives significant performance regressions on all three targets. It gives a particularly large regression on cactuBSSN_r in FP2017 and large regressions on most PlaidML benchmarks.

The ACO algorithm with the base setting produces a geometric-mean improvement of 1.13% on Intel, 1.25% on ARM, and 7.14% on the AMD GPU. The performance gains achieved using the ACO and the B&B algorithms on the GPU target are much more significant than the performance gains achieved on the CPU target. This is attributed to the fact that the impact of RP and ILP on GPU

Table 12. FP2017 on ARM Percentage Increase in Execution Speed Relative to LLVM's Default Scheduler

BENCHMARK	B&B	ACO	CP
503.bwaves_r	1.97%	1.64%	1.31%
507.cactuBSSN_r	4.28%	7.07%	-30.41%
508.namd_r	1.72%	-1.23%	0.98%
510.parest_r	3.91%	3.01%	4.06%
511.povray_r	1.03%	2.27%	1.24%
519.lbm_r	1.13%	0.19%	0.94%
526.blender_r	4.66%	2.76%	6.74%
527.cam4_r	0.79%	0.59%	0.99%
538.imagick_r	-0.83%	-2.08%	-1.66%
544.nab_r	0.00%	0.20%	0.59%
549.fotonik3d_r	0.00%	0.00%	0.00%
554.roms_r	1.50%	0.86%	0.21%
Geo-mean	**1.67%**	**1.25%**	**-1.75%**



performance is higher than their impact on CPU performance. RP determines the GPU occupancy, and increasing occupancy generally has a much higher impact on performance than reducing spilling. Reducing spilling will have a high impact on performance only if a scheduling algorithm makes substantial reductions in spilling in a sufficiently hot region in the code. ILP also has a higher impact on GPU performance, because the GPU does not perform out-of-order execution within a thread, while the Intel and ARM processors used in our evaluation are out-of-order processors.

With the many-ant setting, the performance gain on the AMD GPU increases to 7.66%, compared to 7.14% with the base setting. Although the difference in the geometric-mean gain between the many-ant setting and the base setting is limited, the differences on some individual benchmarks are significant. The many-ant setting gives worse results than the base setting on three benchmarks, which is attributed to negative side effects on unmodeled factors, such as caching and hardware details.

In the previous section, it was shown that the ACO algorithm gives better average occupancy than the B&B algorithm while the B&B algorithm gives a much better average schedule length. The results in Table 13 show that the execution-time performance of the ACO algorithm is slightly better than the B&B algorithm even with the base setting. This is consistent with the fact that occupancy has a higher impact on performance than ILP.

It is important to note here that ACO and B&B are intelligent search algorithms that consider multiple solutions, while the LLVM and AMD algorithms are greedy algorithms. ACO iteratively finds solutions that give better values of an explicit cost function, while a greedy algorithm

Table 13. PlaidML on AMD GPU Percentage Increase in Execution Speed Relative to AMD's Default Scheduler

BENCHMARK	B&B	ACO base	ACO many ants	CP
densenet121	3.56%	4.05%	5.37%	-12.13%
densenet169	3.85%	5.31%	6.52%	-14.80%
densenet201	4.05%	5.06%	5.82%	-15.95%
inception_resnet_v2	16.30%	18.83%	18.79%	-2.42%
inception_v3	15.42%	17.67%	17.29%	8.44%
Mobilenet	3.68%	5.98%	3.92%	-12.69%
nasnet_large	4.11%	6.13%	6.56%	1.93%
nasnet_mobile	4.25%	2.55%	5.74%	-0.51%
resnet50	6.02%	6.09%	7.13%	-12.90%
vgg16	7.35%	8.16%	8.18%	-11.12%
vgg19	6.77%	7.97%	8.35%	-12.00%
Xception	3.90%	4.65%	4.69%	-5.05%
imdb_lstm	1.60%	1.80%	2.43%	1.83%
Geo-mean	**6.13%**	**7.14%**	**7.66%**	**-7.02%**

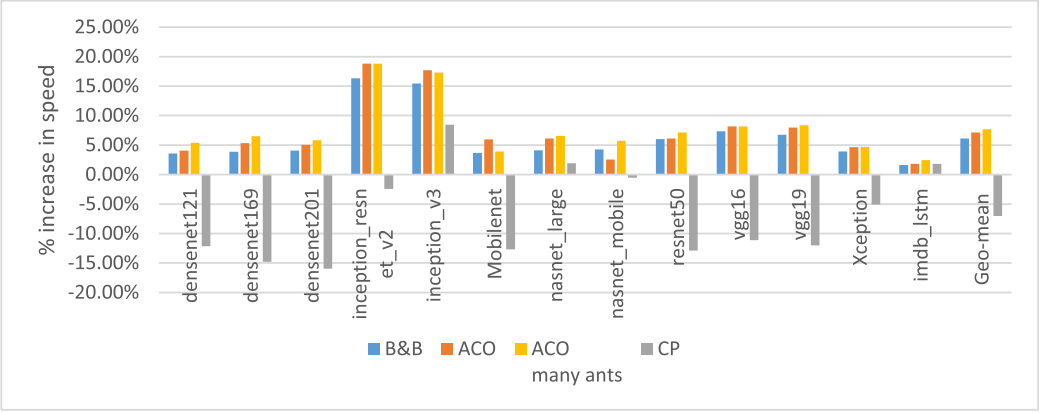


Table 14. Compile Times in Seconds

Benchmarks	LLVM	B&B	ACO	ACO many ants
X86 FP2017 (hot only)	1055	2040	1526	N/A
ARM FP2017 (hot only)	1016	2014	2565	N/A
AMD GPU PlaidML	256	496	588	37,239

constructs a single solution by adding one instruction at a time based on certain priority schemes. If a greedy algorithm makes a bad decision, it does not backtrack to consider other options. The results in this section show that it is hard to optimize two conflicting objectives (RP and ILP) using a greedy algorithm like the LLVM and the AMD algorithms, while ACO and B&B can balance these two conflicting objectives, because they are guided by an explicit cost function and consider multiple solutions.

5.7 Compile Times

Table 14 shows the total compile times taken to generate the executables used to produce the results reported in the previous subsection. Recall that for the CPU targets, ACO and B&B were

applied only to the hot functions in FP2017, while for the GPU targets, ACO and B&B were applied to all the GPU kernels in each benchmark. The table also shows the compile times for the base compilers.

The results in the table show that both the ACO algorithm and the B&B algorithm increase the compile time by significant amounts. However, these compile times are still reasonable for such computationally expensive algorithms, given that the LLVM and the AMD scheduling algorithms are greedy algorithms. The settings of the B&B and ACO algorithm were chosen to give roughly the same compile times. Because each algorithm has a different nature, it is practically impossible to find a combination of settings that gives exactly the same compile time for both algorithms. The different nature of each technique requires a different kind of termination condition. The termination condition for B&B is either completing the exhaustive search or using a certain time budget, whichever happens first. On the other hand, the termination condition for ACO is performing a certain number of iterations without making any further improvement. It would not be reasonable to use such a termination condition for B&B, because B&B is always guaranteed to explore new solutions. Similarly, it would not be reasonable to use a time limit as a termination condition for ACO, because ACO does not prove optimality. So, ACO may find an optimal solution within a small fraction of the time limit, and then spend the rest of the allowed time repeatedly exploring the same solutions. Given these different natures, it is not possible to find a setting that ensures that both algorithms use the same amount of time to process a given set of programs.

In spite of the difficulty of fairly ensuring that both algorithms are given the same amount of time, the results show that each algorithm has an advantage over the other. The results in Table 4 show that B&B can compute provably optimal solutions, while ACO computes better solutions than B&B for thousands of instances that B&B times out on. Therefore, both algorithms are worth further exploration in the future. In future work, it would be interesting to explore a hybrid algorithm that combines the advantages of both B&B and ACO.

Finally, we note that the compile times reported in the Table 14 should be viewed as upper bounds, because our current implementation is a research prototype that involves significant overhead.

6 CONCLUSIONS AND FUTURE WORK

This paper presents an ACO algorithm for the register-pressure-aware instruction scheduling problem, based on the ACO technique proposed by Gambardella and Dorigo [2000] for solving the SOP. The proposed ACO algorithm outperforms LLVM's algorithm on x86 and ARM and outperforms AMD's algorithm on the AMD GPU. It gives geometric-mean improvements of 1.13% and 1.25% on x86 and ARM, respectively, relative to LLVM's scheduler and 7.14% relative to AMD's scheduler on the AMD GPU. The improvement on the GPU target is much more substantial, because, for the reasons explained in the paper, both RP and ILP have a higher impact on GPU performance than on CPU performance.

The proposed ACO algorithm is compared to a B&B algorithm. Each algorithm has its own advantages. The B&B algorithm has the advantage of producing exact solutions when it terminates within the given time limit, while the ACO algorithm has the advantage of producing better solutions than the B&B algorithm for many of the large instances that the B&B algorithm times out on. In terms of execution time, the two algorithms give approximately the same results with the settings used in our experiments. Both algorithms are worth further investigation in future work.

In the future, we plan on developing parallel versions of both algorithms and on exploring a hybrid algorithm that combines the strengths of both algorithms. Furthermore, we are currently exploring an RP-aware version of the graph transformations proposed by Heffernan et al. [2005, 2006] to tighten the solution space before invoking a B&B or an ACO scheduling algorithm.

ACKNOWLEDGMENTS

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

I. Alaya, C. Solnon, and K. Ghédira. 2007. Ant colony optimization for multi-objective optimization problems. In *Proc. 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, Patras, Greece, 1 (2007), 450–457.

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

K. Cooper and L. Torczon. 2011. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition (Feb 2011).

T. C. de Souza Xavier and A. F. da Silva. 2018. Exploration of compiler optimization sequences using a hybrid approach. *Computing and Informatics* 37, 1 (2018), 165–185.

L. Domagala, D. Van Amstel, F. Rastello, and P. Sadayappan. 2016. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proc. International Conference on Compiler Construction*.

M. Dorigo and L. M. Gambardella. 1997. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation* 1, 1 (Apr 1997), 53–66.

M. Dorigo. 1992. *Optimization, Learning and Natural Algorithms (in Italian). Ph.D. thesis*, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy (1992).

L. Escudero. 1988. An inexact algorithm for the sequential ordering problem. *European Journal of Operational Research* 37, 2 (November 1988), 236–249.

F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. 2010. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 6 (June 2010), 911–924.

L. M. Gambardella, É. Taillard, and G. Agazzi. 1999. MACS-VRPTW: A multiple ant colony system for vehicle routing problems with time windows. *New Ideas in Optimization*.

L. M. Gambardella and M. Dorigo. 2000. An ant colony system hybridized with a new local search for the sequential ordering problem. *INFORMS Journal on Computing* 12, 3 (2000).

J. Goodman and W. Hsu. 1988. Code scheduling and register allocation in large basic blocks. In *Proc. Int'l Conf. 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. Computers* 52, 1 (2003), 4–20.

M. Gravel, W. L. Price, and C. Gagné. 2002. Scheduling continuous casting of aluminum using a multiple objective ant colony optimization metaheuristic. *European Journal of Operational Research* 143, 1 (2002), 218–229.

M. Heffernan and K. Wilken. 2005. Data-dependency graph transformations for instruction scheduling. *J. of Scheduling* 8, 5, 427–451.

M. Heffernan, K. Wilken, and G. Shobaki. 2006. Data-dependency graph transformations for superblock scheduling. In *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA. 77–88.

X. M. Hu, J. Zhang, J. Xiao, and Y. Li. 2008. Protein folding in hydrophobic-polar lattice model: A flexible ant colony optimization approach. *Protein and Peptide Letters* 15, 5 (2008), 469–477.

M. L. Infante and T. Stützle. 2010. The impact of design choices of multiobjective ant colony optimization algorithms on performance: An experimental study on the biobjective TSP. In *Proc. of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Portland, OR, USA.

A. Jevtić, J. Quintanilla-Dominguez, M. G. Cortina-Januchs, and D. Andina. 2009. Edge detection using ant colony search algorithm and multiscale contrast enhancement. *IEEE International Conference on Systems, Man and Cybernetics*, 2009. 2193–2198.

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. on Design Automation of Electronic Systems* 1, 4 (1996), 443–456.

C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. International Symposium on Code Generation and Optimization (CGO 2004)*.

G. Leguizamón and C. A. Coello Coello. 2010. Multi-objective ant colony optimization: A taxonomy and review of approaches. Integration of swarm intelligence and artificial neural network. 2011, 67–94.

C. Lintzmayer, M. H. Mulati, and A. F. da Silva. 2012. Register allocation with graph coloring by ant colony optimization. In *Proc. 30th International Conference of the Chilean Computer Science Society*.

R. C. Lozano. 2018. *Constraint-based register allocation and instruction scheduling*. Doctoral thesis, KTH Royal Institute of Technology, 2018.

R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. on Programming Languages and Systems* 41, 3 (2019), Article 17.

R. C. Lozano and Christian Schulte. 2019. Survey on combinatorial register allocation and instruction scheduling. *ACM Computing Surveys* 52, 3 (2019), Article 62.

V. Makarov. 2013. Mechanism for performing instruction scheduling based on register pressure sensitivity. U.S. Patent No. 8,549,508. 2013.

A. Malik. 2008. *Constraint Programming Techniques for Optimal Instruction Scheduling. Ph.D thesis*, University of Waterloo, 2008.

D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, and B. Baesens. 2007. Classification with ant colony optimization. *IEEE Transactions on Evolutionary Computation* 11, 5 (2007), 651–665.

C. Mu, J. Zhang, Y. Liu, R. Qu, and T. Huang. 2019. Multi-objective ant colony optimization algorithm based on decomposition for community detection in complex networks. *Soft Computing* 23, 23 (2019), 12683–12709.

J. Ning, C. Zhang, P. Sun, and Y. Feng. 2019. Comparative study of ant colony algorithms for multi-objective optimization. *Information* 10, 1 (2019).

Perf (Performance analysis tool on Linux). https://en.wikipedia.org/wiki/Perf_(Linux).

PlaidML machine learning benchmarks. https://github.com/plaidml/plaidbench#intel-corporation-machine-learning-benchmarks.

P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L. Pouchet, A. Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proc. 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2018)*.

R. Sethi and J. D. Ullman. 1970. The generation of optimal code for arithmetic expressions. *J. of the ACM* (1970), 715–728.

G. Shobaki, M. Shawabkeh, and N. E. A. Rmaileh. 2013. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Trans. Archit. Code Optim.* 10, 3, Article 14 (Sep 2013).

G. Shobaki, L. Sakka, N. E. A. Rmaileh, and H. Al-Hamash. 2015. Experimental evaluation of various register-pressure-reduction heuristics. *Softw. Pract. Exper.* 45, 11 (Nov 2015), 1497–1517.

G. Shobaki, A. Kerbow, C. Pulido, and W. Dobson. 2019. Exploring an alternative cost function for register-pressure-aware instruction scheduling. *ACM Trans. Archit. Code Optim.* 16, 1, Article 1 (March 2019).

G. Shobaki, A. Kerbow, and S. Mekhanoshin. 2020. Optimizing occupancy and ILP on the GPU using a combinatorial approach. In *Proc. International Symposium on Code Generation and Optimization (CGO 2020)*.

R. Skinderowicz. 2015. Population-Based ant colony optimization for sequential ordering problem. *Computational Collective Intelligence*. Springer, Cham, 99–109.

SPEC CPU 2017. https://www.spec.org/cpu2017/.

S. Touati. 2005. Register saturation in instruction-level parallelism. *Intl. Journal of Parallel Prog.* 33, 4 (2005), 393–449.

G. Wang, W. Gong, B. DeRenzi, and R. Kastner. 2007. Ant colony optimizations for resource and timing constrained operation scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System* 26, 6 (2007).

Received May 2021; revised November 2021; accepted December 2021