Hybrid Register Allocation with Spill Cost and Pattern Guided Optimization

Yongwon Shin and Hyojin Sung

Pohang University of Science and Technology (POSTECH), Pohang, South Korea,
{ywshin,hsung}@postech.ac.kr

Abstract. Modern compilers have relied on various best-effort heuristics to solve the register allocation problem due to its high computation complexity. A "greedy" algorithm that performs a scan of prioritized live intervals for allocation followed by interval splits and spills is one of the widely used register allocation mechanisms with consistent performance and low compile-time overheads. However, its live interval splitting heuristics suffer from making sub-optimal decisions for scenarios hard to predict, and recent effort to remedy the issue is not free from unintended side effects with performance degradation. In this paper, we propose Greedy-SO, a greedy register allocator with a spill cost and pattern guided optimization that systematically addresses inherent sub-optimalities in live-interval splitting by avoiding them for codes whose performance are more likely to be impacted by sub-optimal decisions. Greedy-SO identifies functions with such code patterns, precisely models the spill cost for them during the greedy allocation process, and switches to an alternate allocator without interval splitting when the spill cost starts to deteriorate. Our hybrid register allocator improves the performance of target benchmarks up to 16.1% (7.3% on average) with a low compilation overhead, while not impacting non-target benchmarks at all.

1 Introduction

Registers are scarce and valuable hardware resources whose software-managed utilization can significantly impact code performance. Modern compilers have solved the problem of register allocation, i.e., mapping infinite registers to limited architectural registers, by finding an optimal solution that maximizes utilization and minimizes memory spills [6, 14].

As it is a known NP-complete problem to find an optimal register allocation for realistic codes [11], various best-effort heuristics have been proposed, including graph-coloring based algorithms [8, 9], linear scan algorithms [22], and their many variants [17, 26]. Graph-coloring based algorithms adapt k-coloring heuristics to assign colors, i.e., registers, for conflicting live intervals represented as connected graph nodes, while priority-based algorithms exploit program information to determine the allocation order.

Register allocators in modern compilers implement a version of the above allocation algorithm(s) with several other heuristics for register spill and live interval coalescing/splitting decisions. For example, the “greedy” algorithm in the LLVM compiler coalesces live intervals first to reduce conflicts and allocates registers for them in their priorities computed from variable types and access patterns. Then it splits remaining unassigned intervals into pieces and spills occupied registers until the allocation is completed.

While these heuristics are carefully designed and fine-tuned with expert knowledge and benchmark testing, they are bound to suboptimal decisions due to approximate modeling of the problem. Especially the live-interval splitting heuristic involves complicated logic to determine which live interval to split and where in an interval as well, which can lead to widely varying code thus performance. Recent work [27] tried to improve the heuristic, but it introduces performance degradation in unintended cases as a result of unpredictable chains of interactions between the changed heuristic and the rest of the register allocator.

Our key insight is that the heuristic itself is not a problem, but the way register allocation is performed *without* a systematic cost model that estimates the profitability of given heuristics for the code and adaptively selects allocation approaches. While previous work [10, 16] enabled hybrid allocation using different allocators per function or code segment, we focus mainly on structuring *the internal phases of the allocation process* to use optimal heuristics based on modeled cost. This cost-guided optimization will enable more effective register allocation while minimizing heuristic-engineering effort and negative performance impact.

Thus, we propose Greedy-SO (**S**plit **O**ptimization), a hybrid register allocator with spill cost and pattern guided optimization for live-interval splitting logic in the LLVM greedy allocator. Our allocator uses improved spill cost modeling to detect if and when suboptimal splitting decisions occur in the greedy allocator. Then, we switch to an alternative register allocator to finish the rest of the allocation process without splitting. We use empirically identified code patterns and thresholds to determine if this alternative allocation path will provide performance gains and apply only when it is predicted to. The target code patterns turn out to have heavy computation in large loop bodies, which supports our assumption that such codes with high register pressure are more likely to suffer from the suboptimal splitting logic and benefit from our solution.

The evaluation showed that Greedy-SO improves the performance of five benchmarks in LLVM test suite by 7.3% on average (up to 16.1%) *without impacting other benchmarks* on Intel CPU. Since these benchmarks can be used as building blocks for larger applications, the potential performance gain will be significant at an application level. We believe that this paper shows the promising potential towards systematic optimization-driven back-end code generation.

In the rest of the paper, motivating insights and preliminary analysis are presented in Section 2 and 3. Then we describe the design and implementation of Greedy-SO in detail in section 4. Section 5 and 6 describe experimental setup and results for Greedy-SO and other allocators. The paper wraps up with related work and a conclusion.

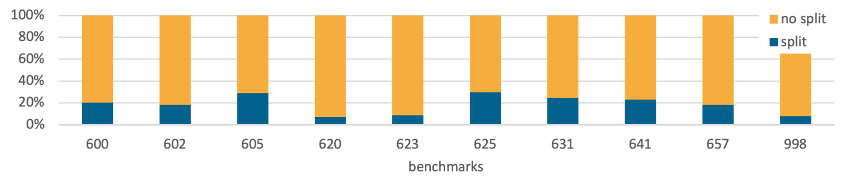

Fig. 1: The ratio of functions entering the split phase in SPEC2017 intspeed

2 Background and Challenges

We made the following observations that reveal the challenges of existing register allocators that guided our design in Section 3 and 4.
Observation 1. Codes compiled with the greedy allocator are subject to high performance variability from live interval splitting heuristics. The LLVM greedy register allocator consists of three main phases. As a preliminary step, it computes live intervals of virtual registers and assigns priorities using program-based criteria, e.g., giving a higher priority to global variables and variables within a loop. The coalescing phase combines related live intervals through copies to reduce register pressure. In the allocation phase, the live intervals sorted in a priority queue are assigned to an available register. The register allocation may finish here if no registers remain to be allocated or move on to the final split/spill phase, which splits allocated live intervals into smaller ones to reassign them or spills live intervals to the memory to make room.

All the heuristics in these phases closely interact to generate final allocation, and precisely analyzing its workings is likely to be fruitless. However, we observe that the live interval splitting heuristics have a much larger decision space than the others, e.g., whether to split or spill, which live interval to split, where in a live interval to split and at which level (region, block, or local), and in how many sub-intervals to create, which can lead to higher performance variability caused by heuristic design. Considering the fact that more than 25% of the functions in SPEC2017 have to go through the split/spill phase when compiled with default options for Intel CPU as shown in Figure 1, optimizing the heuristics in the split/spill phase can bring out performance gain for a wide range of codes on dominant CPU platforms.
Observation 2. Optimizing heuristics for specific cases often introduces unexpected performance degradation in others. Carefully designed heuristics are often updated to handle corner cases where they perform pathologically bad under different circumstances. However, once heuristics are mature, adjusting them to fix specific cases without negative side effects is very difficult and requires significant engineering and testing effort without a guarantee. For example, recent work [27] identified and addressed an issue of not considering local interference as weights in the region-level splitting heuristic. While it works well for target testcases, it randomly introduces performance degradation as the changed heuristic can cause unexpected side effects with the rest of the register allocator as reported by the community and our evaluation.

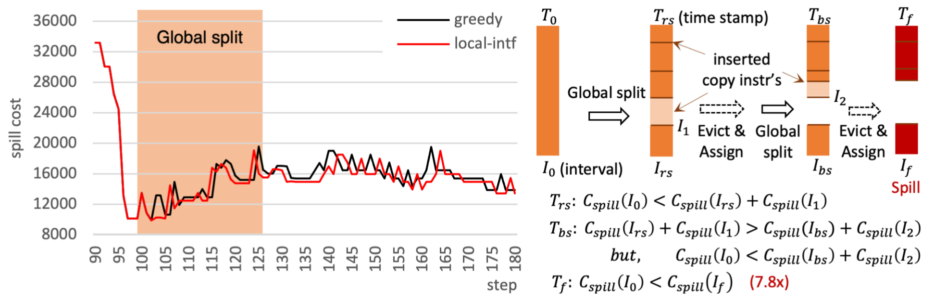

Fig. 2: Cost tracking graph (left) and a case of suboptimal splitting (right) for bicubic interpolation kernel [4]. The orange box in the left indicates where region and block splits happen.

This observation calls for detailed cost modeling of the heuristics to reveal potential suboptimal decisions and adaptively apply optimizations based on cost prediction.

3 Preliminary Analysis

To understand how heuristic decisions impact the potential cost of register allocation, we modeled the register spill cost for each live interval as follows:

$$C_{load} = \sum_{u \in U} B_{freq}(u), \;\; C_{save} = B_{freq}(d) \tag{1}$$

$$C_{spill} = R * (C_{load} + C_{save}) \tag{2}$$

$$C_{total\_spill} = \sum_{i \in I} C_{spill}(i) \tag{3}$$

C_{load} and C_{save} estimate runtime memory load and save costs by summing up block frequencies for each use (u) and def (d) in a given live interval. A live interval in LLVM is defined per virtual register whose definition (d) is unique per interval in the SSA form [23]. C_{spill} is computed as a sum of the two costs discounted by a rematerialization ratio (0.5 if all uses are rematerializable). By adding up C_{spill} of the spilled intervals and that of intervals in the priority queue, C_{total_spill} conservatively estimates the memory spill cost of the current register allocation state assuming the worse cases where all remaining intervals are spilled.

Similar forms of spill costs were used to evaluate the efficiency of register allocators in prior work [8, 24], but they focus on computing the final spill cost with completed register allocation. We designed this spill cost to estimate the cost of register-to-memory spills at a given point of the register allocation process. This spill cost is also different from the weight used in the greedy allocator for many heuristic decisions including splits and spills. The greedy allocator normalizes the weight with live-interval length so that it can prefer short live intervals for allocation and long live intervals for splitting [21]. We do not factor live interval lengths into our spill cost because it is designed to keep track of "after-the-fact" states of a heuristic decision.

No live intervals are assigned yet at the beginning of register allocation, so we start at the highest possible spill cost assuming that they will be all spilled into the memory and should be loaded to registers for execution. As live intervals are allocated to registers, the spill cost decreases. Live interval splitting will increase the spill cost by the sum of block frequency for each inserted instruction, then decrease it by assigning a split live interval to a register. Register spills increase the spill cost by the cost of the live interval.

Ideally, the spill cost should continuously go down throughout the register allocation process, producing the minimal cost at the end. Splits create spikes but the spill cost after a split should be lower than before. However, our preliminary experiments with LLVM test suite benchmarks reveal many non-ideal cases. In Figure 2, we can see that the final spill costs obtained by the original greedy allocator and an optimized version [27] are both higher than the minimal cost obtained early in the split phase. A closer look at a suboptimal splitting case as shown on the right side of Figure 2 reveals how the final spill cost at T_f ends up higher than the spill cost before splits at T_0. Split attempts at T_{rs} and T_{bs} allow I_1 and I_2 to be allocated reducing the total cost, but the costs for inserting copy instructions are higher than that. This suboptimal behavior is challenging to predict with iterative eviction-assignment chains.

Our preliminary analysis revealed the following.

1. The *global-split* heuristic, which considers live intervals spanning multiple basic blocks for splitting candidates, makes suboptimal decisions, thus causing the overall spill cost to go up after a split. This issue is partially addressed by [27].
2. Cost tracking graphs for many benchmarks show suboptimalities with the *block-split* heuristic as well. This is because the current greedy allocator does not consider the local interference for block splitting, and the issue has been reported or addressed by prior work.
3. [27] can make different suboptimal decisions than the baseline greedy allocator leading to an even higher spill cost at the end, thus worse performance (details in Section 6), which shows it cannot be trusted to improve the allocation quality in general.

The analysis result, combined with the observations in Section 2, guides our systematic cost-driven design of register allocator in the next section that focuses on addressing suboptimalities at an allocation strategy level rather than an individual heuristic design level.

4 Design and Implementation

Figure 3 shows the compilation flow of the Greedy-SO register allocator with three main components. It consists of two cost models, **code pattern rec-**

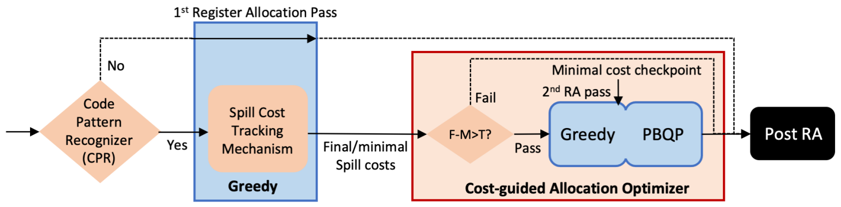
1st Register Allocation Pass
No
Code Pattern Recognizer (CPR)
Yes
Spill Cost Tracking Mechanism
Greedy
Final/minimal Spill costs
Minimal cost checkpoint
2nd RA pass
Fail
F-M>T?
Pass
Greedy
PBQP
Cost-guided Allocation Optimizer
Post RA

Fig. 3: The Overview of Greedy-SO

ognizer and **spill cost tracking mechanism**, and **cost-guided allocation optimizer** that selectively applies hybrid register allocation based on the combined cost. Code pattern recognizer serves as a counter-based filter to determine if the compiled code is an optimization candidate that requires detailed spill code tracking. Second, for identified candidates, the spill cost tracking mechanism computes the spill cost as proposed in Section 3 during register allocation to detect potential suboptimalities. The two costs, code statistics and spill cost, are evaluated sequentially to eliminate spill cost tracking overheads for non-target codes. Lastly, after the first register allocation process, the cost-guided allocation optimizer examines the minimum spill cost (M) and the final spill cost (F) with a threshold (T), and if the condition is met, reverts the allocation result and executes an alternate allocation path from when the minimal spill cost is reached. The following sections describe the design and implementation of each component in detail.

4.1 Code Pattern Recognizer

The code pattern recognizer (CPR) collects counter-based code statistics by recursively traversing all loops in a function and compute conditions that represent the target code pattern of Greedy-SO register allocator. The goal of the CPR is to focus on optimizing functions for which suboptimal splits are likely to be translated into performance gains. In other words, it filters out functions that are too sensitive to architectural noises such as code alignment and instruction cache misses [2]. For example, we observed that when most of the innermost loops are small or instructions in small inner loops dominate the total number of instructions, non-deterministic side effects caused by changes in register allocation often hide the performance benefit of Greedy-SO. Thus, CPR checks if any of the following conditions holds for filtering.

1. Small loops only. CPR filters out functions whose basic blocks in loops have all less than C_{small} (= 10) instructions. While small loops can go through the split/spill phase, they are susceptible to microarchitectural noises and not likely to have stable performance gains from our optimization.

$$F_{small\_loop} = \begin{cases} true & \text{if all basic blocks has less than } C_{small} \text{ instructions} \\ & \text{in all loops} \\ false & \text{otherwise} \end{cases} \tag{4}$$

2. A lot of small innermost loops. CPR applies relaxed filtering criteria with an increased threshold for C_{small} (= 15) and tries to detect functions that have lots of innermost loops whose majority are small. This condition targets functions with many function calls inlined by small loops.

$$F_{small\_inline} = \begin{cases} true \;\; \text{if } (L_{inner} > c_1) \wedge (ratio\_small > c_2) \\ false \; \text{otherwise} \end{cases} \tag{5}$$

where $ratio\_small = L_{small\_inner}/L_{inner}, c_1 = 30, c_2 = 0.85$.
3. Middle-sized loops. CPR further relaxes the criteria to detect middle-sizes loops, decreasing the *ratio_small* threshold. We observed that even with a moderately high *ratio_small* value, if a function also has many instructions in non-small and non-innermost loop bodies, Greedy-SO can tolerate architectural performance jitters. This condition is checked by comparing the ratio of I_{middle_loop} (# of instructions in non-innermost or non-small loops) and I_{inner} (# of instructions in small innermost loops).

$$f_1 = \begin{cases} true \;\; \text{if } (ratio\_small > c_3) \wedge (I_{middle\_loop} < I_{inner} * c_4) \\ false \; \text{otherwise} \end{cases} \tag{6}$$

$$f_2 = \begin{cases} true \;\; \text{if } (ratio\_small > c_5) \wedge (I_{middle\_loop} < I_{inner} * c_6) \\ false \; \text{otherwise} \end{cases} \tag{7}$$

$$F_{middle\_loop} = f_1 \vee f_2 \tag{8}$$

where f_1 uses $c_3 = 0.5$ and $c_4 = 3.5$ while f_2 uses more relaxed parameters than f_1: $c_5 = 0.4$ and $c_6 = 4.2$.

Finally, CPR filters out a function if any of the previous conditions is true.

$$F = F_{small\_loop} \vee F_{small\_inline} \vee F_{middle\_loop} \tag{9}$$

The filtering conditions and parameters(C_{small}, c_1 to c_6) are hand-tuned through experiments on Intel and AMD machines Adopting a more systematic and learning-based approach such as rule induction [13] can help improve the heuristic design, and it is part of our future work.

Figure 4 shows examples of a target function and a non-target function. The function on the left has an innermost loop with many array accesses on distinct locations (in red). This function will have to perform numerous live interval splits and spills due to high register pressure. Also, improved register allocation for the large inner loops will have high performance impact. On the other hand, the function on the right has only one loop nest with a small innermost loop (in green). Even if the spill cost modeling reveals suboptimalities, non-deterministic architectural behaviors could easily offset performance gain by Greedy-SO.

4.2 Spill Cost Tracking Mechanism

We implemented the spill code computation and tracking in the LLVM greedy allocator as described in Section 3. C_{spill} is computed for every live interval when

```
float jacobi(...) {...
for(n=0 ; n<nn ; n++){
  gosa = 0.0;
  for(i=1 ; i<imax; i++)
    for(j=1 ; j<jmax ; j++)
      for(k=1 ; k<kmax ; k++) {
    s0= MR(a,0,i,j,k)*MR(p,0,i+1,j,  k)
      + MR(a,1,i,j,k)*MR(p,0,i,  j+1,k)
      + MR(a,2,i,j,k)*MR(p,0,i,  j,  k+1)
      + MR(b,0,i,j,k)
       *( MR(p,0,i+1,j+1,k) - MR(p,0,i+1,j-1,k)
        - MR(p,0,i-1,j+1,k) + MR(p,0,i-1,j-1,k) )
      + MR(b,1,i,j,k)
       *( MR(p,0,i,j+1,k+1) - MR(p,0,i,j-1,k+1)
       - MR(p,0,i,j+1,k-1) + MR(p,0,i,j-1,k-1) )
      + MR(b,2,i,j,k)
       *( MR(p,0,i+1,j,k+1) - MR(p,0,i-1,j,k+1)
        - MR(p,0,i+1,j,k-1) + MR(p,0,i-1,j,k-1) )
      + MR(c,0,i,j,k) * MR(p,0,i-1,j,  k)
      + MR(c,1,i,j,k) * MR(p,0,i,  j-1,k)
      + MR(c,2,i,j,k) * MR(p,0,i,  j,  k-1)
      + MR(wrk1,0,i,j,k);
  ... }
```

```
int dfilter(...) ... {
  ...
  for (i=0; i<high; i++)
    for (j=0; j<larg; j++) {
        for (l=-(tm_g/2); l<=(tm_g/2); l++) {
if ((j+l) < 0) nv = (float) image[i*larg];
else if ((j+l) >= larg) nv = (float) image[((i+1)*larg)-1];
else nv = (float) image[(i*larg)+j+l];
*d = (nv * g[(tm_g/2)-l]) + *d;
        }
        d++;
    }
  ...
  d2 = (float *) calloc(nc*nr, FWS);
  if (!d2) {
    sprintf(err,"Out of memory");
    return(1);
  }
  ...
```

▲ Non-target function with low register pressure in the loop

◀ Target function of high register pressure and local interferences in the loop

Fig. 4: Example target and non-target functions for Greedy-SO

created and pushed into a priority queue and recomputed when a new interval is generated from live interval splitting. C_{total_spill} is computed at the following tracking points in the register allocation process: (1) when a live interval is enqueued into the priority queue, (2) when a live interval is dequeued from the priority queue, and (3) when a spill occurs.

For example, when a split happens, an interval is dequeued from the priority queue at t_0 and split into smaller intervals. The total spill cost does not change for the dequeue action since we conservatively count intervals in the queue as potential spills. Then the split intervals are enqueued back to the priority queue for allocation at t_1. At this point, the total cost will go up as a split introduces additional copy instructions at interval boundaries. When split intervals are dequeued and allocated later at t_2, successful allocation will eventually result in a lower spill cost than the spill cost at t_0 as their costs are all deducted from the total spill cost. However, suboptimal splitting decisions that cause spills for split intervals will not be able to deduct their costs and the total spill cost will stay higher than the cost at t_0. The key idea behind Greedy-SO is that if the total spill cost gets much worse after reaching the minimum earlier in the process, stopping when the minimum is reached enables us to avoid suboptimal splitting decisions afterward.

We keep track of the minimal total spill cost by comparing a newly computed total spill cost with the current minimal total spill cost at the above tracking points, then make a checkpoint for when a new minimum appears. A checkpoint is stored as the number of "dequeue" events of the priority queue. Greedy-SO uses this checkpoint to determine when an alternate fall-back allocator takes over the allocation process.

4.3 Putting It All Together: Cost-Guided Allocation Optimizer

After the first register allocation with the spill cost tracking enabled, the cost-guided allocation optimizer compares the final spill cost ($cost_f$) and the minimum spill cost ($cost_m$) and evaluates the following condition to decide whether or not to proceed to hybrid register allocation:

$$\begin{aligned}&(cost_f \geq 100 \ \wedge \ cost_m < cost_f * 0.9) \ \vee \\ &(50 \leq cost_f < 100 \ \wedge \ cost_m < cost_f * 0.8)\end{aligned} \tag{10}$$

The condition filter outs codes with a spill cost lower than 50 since such functions should be very small and not worth optimizing. Conversely, the condition favors codes with a high spill cost (> 100) which are likely to have a high performance impact by giving them a weak threshold. A function is recognized as a target function of Greedy-SO optimization by passing this test.

Then Greedy-SO starts the second register allocation pass as the greedy allocator based on a snapshot of machine functions created before the first allocation pass (details in Section 5). When Greedy-SO arrives at the saved checkpoint, it passes the analysis result and current allocation information to a fall-back register allocator, Partitioned Boolean Quadratic Programming (PBQP) [18, 25] or LLVM Basic allocator. PBQP constructs a cost matrix for each pair of virtual registers and records spill cost, register aliasing information, and optional coalescing profitability. Then it tries a PBQP solver to get a solution. If not solved, registers with the lowest spill cost are spilled, and the PBQP allocator repeats the process until all the constraints are satisfied. Greedy-SO passes a pointer to LLVM spiller (in charge of register spills) to PBQP. Once PBQP takes over, it solves the remaining allocation problem from the checkpoint.

5 Methodology

Compiler implementation. We implemented Greedy-SO in LLVM 13 [19] by modifying the greedy register allocator to follow the compilation flow of Greedy-SO when a compilation option “use-greedy-so” is given. If a function is identified as a target function after the first allocation pass, a modified `FPPassManager` clones the original function at LLVM IR level and replaces all uses of the target function with those of the cloned function. Then, the cloned function goes through the same machine function passes as the original function then the Greedy-SO register allocation pass.

Benchmarks. For experimental results, we evaluated 737 benchmarks from the LLVM test suite [3] except for the ones under “External” and “CTMark” categories, with a performance testing option (`TEST_SUITE_BENCHMARKING_ONLY`). Experiments were repeated ten times with a single thread and default inputs using `llvm-lit` test tool.

Experimental setup. For Greedy-SO, we set the minimal spill cost threshold to trigger hybrid register allocation to 0.1, i.e., the difference between the minimal spill cost and the final spill cost should be more than 10%, and used PBQP register allocator in LLVM whose implementation is based on [18]. We performed a sensitivity study to see how the spill cost threshold and the type of fall-back register allocator affect the performance of Greedy-SO in Section 6.2. The evaluated configurations are as follows:

- *greedy* and *pbqp*: The original LLVM greedy and PBQP allocator in LLVM 13 with default options.
- *local-intf*: *greedy* with "consider-local-interval-cost" option enabled [27].
- *gs* and *gs-basic*: *gs* is Greedy-SO, and *gs-basic* replaces the fall-back register allocator with the LLVM basic register allocator.
- *gs-wop-pbqp*, *gs-wop-pbqp-0.1* and *gs-wop-pbqp-0.2*: Greedy-SO without the code pattern recognizer with varying spill cost threshold. *gs-wop-pbqp* has 0 for the threshold, i.e., it always triggers hybrid register allocation when the final spill cost is not the minimal spill cost, while 0.1 and 0.2 signify the required difference ratio between minimum spill cost and final spill cost for the trigger.
- *gs-wop-basic*, *gs-wop-basic-0.1* and *gs-wop-basic-0.2*: *gs-basic* without the code pattern recognizer with varying spill cost threshold.

We evaluated LLVM test suite on Intel (Xeon Gold 5218 CPU with 375GB RAM), AMD (EPYC 7571 with 64GB RAM), and ARM (Neoverse N1 with 64GB RAM) CPU platforms. Hyperthreading and CPU frequency scaling are disabled for the Intel CPU.

6 Evaluation Result

In our performance evaluation of Greedy-SO register allocator, we focus on showing that Greedy-SO provides improved or comparable performance for target codes identified by the cost models compared to the greedy allocator. We also compared Greedy-SO with previous work [27] and variations of Greedy-SO to show that both the spill cost modeling and pattern-based filtering are crucial to provide consistent performance without degradation.

6.1 Performance Evaluation

Figure 5 and 6 provide execution times of a set of LLVM test suite benchmarks, normalized to the LLVM greedy allocator. We evaluated the entire test suite with these allocators and found out that 97.7% (97.8% in AMD) of the total benchmarks show less 2% of performance variations regardless of the type of register allocator. While 2.3% (2.2% in AMD) of such benchmarks include target functions for Greedy-SO optimization, most of them have a negligible performance impact. Thus, we focus on investigating the benchmarks whose performance is affected by more than 2% (better or worse) with any of the evaluated allocators. In Figure 5 and 6, the benchmarks in the left section are the ones showing more than 2% speedup (or slowdown) with Greedy-SO, while the benchmarks on the

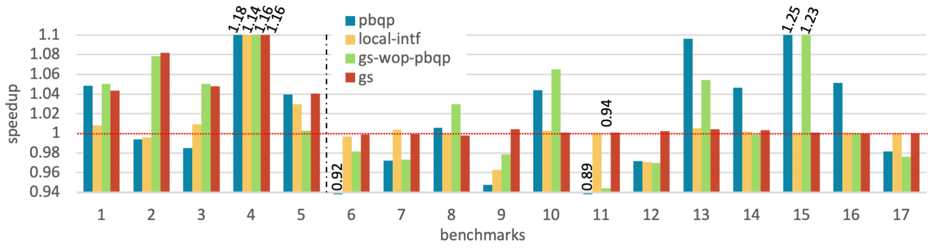

Fig. 5: Performance of LLVM test suite on Intel CPU (normalized to Greedy allocator)

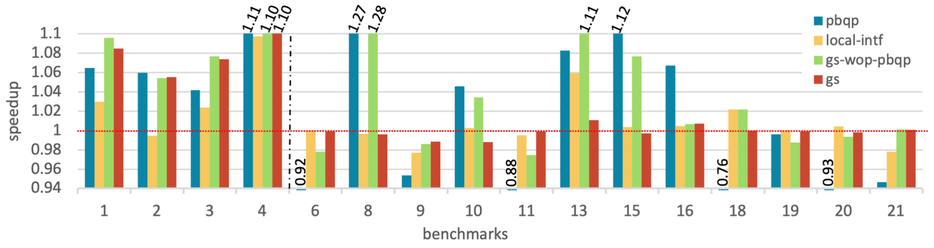

Fig. 6: Performance of LLVM test suite on AMD CPU (normalized to Greedy allocator)

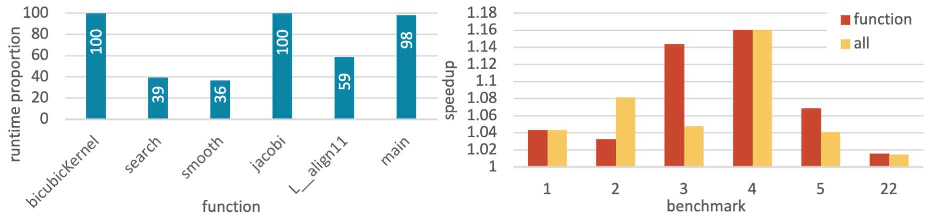

Fig. 7: Comparison of target function performance (*function*) and benchmark performance (*all*) (right) and the runtime proportion of target functions (left).

Table 1: Selected LLVM test suite benchmark list.

Benchmarks					
1	BICUCIB_INTERPOLATION	9	lencod	17	functionobjects
2	aha	10	lambda	18	MemCmp<4, EqZero, None>
3	miniGMG	11	lua	19	SIBsim4
4	himenobmtxpa	12	siod	20	obsequi
5	pairlocalalign	13	PathFinder	21	sqlite3
6	ENERGY_CALC_LAMBDA	14	XSBench	22	dynprog
7	TRAP_INT_RAW	15	yacr2		
8	FIND_FIRST_MIN_LAMBDA	16	ks		

right are affected more than 2% by other evaluated register allocators, but not by Greedy-SO (benchmark information in Table 1).

Target benchmarks. Greedy-SO showed 7.3% and 7.9% (up to 16.1% and 10.1%) average speedup for the target benchmarks on Intel and AMD CPU, respectively. Function-level analysis revealed that functions optimized with Greedy-SO in these benchmarks do contribute mainly to the overall performance gain, as shown in Figure 7 (For benchmark `2`, the overall speedup is greater than the per-function speedup as there are non-target functions impacted by changed code alignment). On a closer look, these functions tend to have loops with large, compute-intensive bodies (some as a result of function inlining) as we modeled with the code pattern in Section 4. Their high register pressure led to a long split/spill process where suboptimal heuristic decisions substantial enough to translate into performance degradation are more likely to occur.

Greedy-SO shows consistently better or comparable performance than all the other evaluated allocators for these benchmarks. While *local-intf* improves greedy for several benchmarks, it does not show any performance improvement with some of the benchmarks where Greedy-SO does (`2` and `3` on Intel CPU and `2` on AMD CPU). These benchmarks suffer from suboptimal heuristic decisions according to our spill cost modeling, but *local-intf* as another splitting heuristic seems to be not able to address them consistently. *gs-wop-pbqp* does not use the code pattern recognizer, so it optimizes a superset of functions included in Greedy-SO. It provides comparable performance to Greedy-SO for most of the benchmark except for `5` for which we suspect *gs-wop-pbqp* introduces performance degradation for non-target functions. This result strongly supports the potential for our hybrid approach as a robust and effective solution for register allocation.

While the performance gain on Intel and AMD CPU vary due to microarchitectural differences, they affect the same set of benchmarks except for `5`. This shows that the code pattern we used can generalize across different hardware.

Non-target benchmarks. The right section in Figure 5 and 6 includes the benchmarks with more than 2% performance jitter with the other register allocators than Greedy-SO. We analyzed these benchmarks to see how effective the combination of the spill cost and code patterns in Greedy-SO is in avoiding potential performance degradation by focusing on profitable cases only. Only 0.63% (0.55% in AMD) of the functions in these benchmarks are optimized by Greedy-SO while the rest does not satisfy both conditions. As a result, Greedy-SO and greedy show little performance difference.

On the other hand, the execution times of the other register allocators are wildly varying without any consistent trend. The performance of *pbqp* and *local-intf* fluctuate from -24% to 28%, and understanding its source is not in the scope of this paper. *gs-wop-pbqp* generously applies hybrid register allocation whenever the spill cost indicates suboptimalities. This produces a mixed result of improving or degrading non-target functions. We found that performance degradation comes from random negative side effects from changed allocation being stronger than the performance gain. We can see that the code pattern recognizer is crucial in avoiding these cases and applying the optimization only when overall performance improvement is expected.

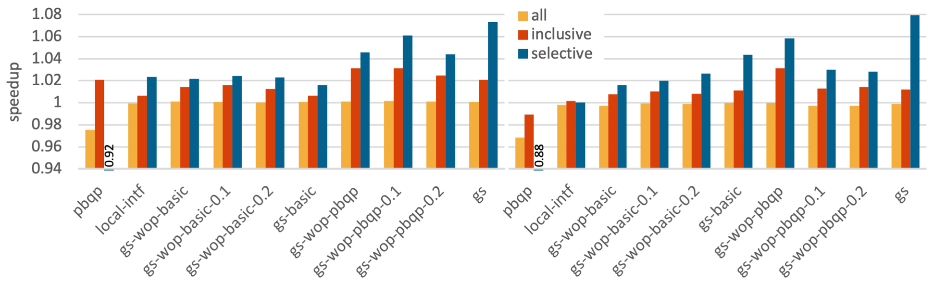

Fig. 8: Sensitivity study for spill cost thresholds and fall-back register type. *all* shows average performance for the entire LLVM test suite (737 total), while *selective* is for the subset of benchmarks with more than 2% performance difference compared the greedy allocator in a given configuration and *inclusive* is for the union of *selective* sets (except for *pbqp* sets; 25 for Intel and 26 for AMD).

As for the benchmarks with performance improvement (`8`, `10`, `13`, `14`, `15` and `16`), we identified two main sources. In some cases, our code pattern is too strong and restrictive to identify marginally profitable codes as target functions (details provided in the sensitivity study in Section 6.2). We also discovered that a significant speedup for `8` comes from the internal workings of microarchitectural features in Intel CPU. When a function has many small innermost loops, its performance gets very sensitive to the code size and alignment. We excluded functions likely to trigger this phenomenon from our code pattern for the stable outcome without performance degradation.

In summary, the evaluation shows that Greedy-SO can provide more efficient register allocation, thus improving performance for most impacted functions by suboptimal splitting heuristics in the LLVM greedy allocator without impacting other functions.

ARM CPU. We did not observe any statistically meaningful performance differences among all evaluated register allocators on ARM CPU. It is due to much lower register pressure in general, with more architectural registers on ARM CPU than X86 CPU's.

6.2 Sensitivity Study

We conducted a sensitivity study with varying spill cost thresholds and types of fall-back register allocator on Intel and AMD CPU, as shown in Figure 8 and 9. Figure 8 presents average execution times of three benchmark sets compiled with ten different configurations as described in Section 5, while Figure 9 shows statistics for benchmarks in the *selective* set.

In terms of spill cost thresholds, both *gs-wop-basic-** and *gs-wop-pbqp-** show mixed results on Intel and AMD CPU. In Figure 9, we can see that green bars consistently shrink as the threshold gets stronger, while red bars stay the same or even bigger. This shows that the spill cost thresholds are not precise or sufficient

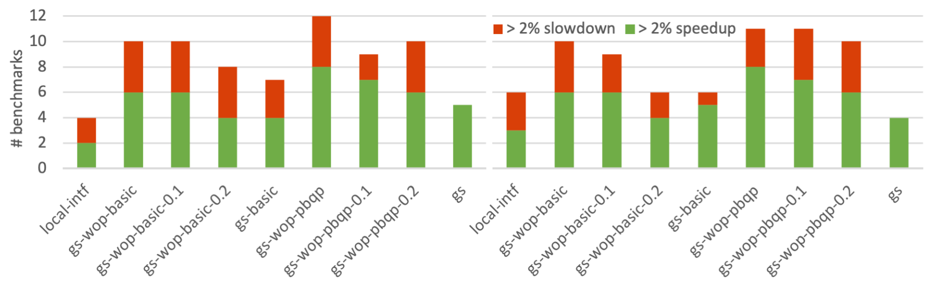

Fig. 9: Function performance of the *selective* benchmark set for configurations with different fall-back register allocators and spill cost threshold.

enough to filter out noisy cases and focus only on promising cases. Comparing Greedy-SO and *gso-wop-** with the spill cost modeling only, we found that Greedy-SO efficiently excludes noisy cases than the versions with thresholds only.

Another important factor is the efficiency of the fall-back register allocator, which determines how much improvement over the existing greedy allocator can be achieved. Figure 8 shows that *gs-wop-pbqp-** and *gs* provide performance comparable to or higher than *gs-wop-basic-** and *gs-basic*. We found out that a conservative and quick eviction algorithm of the basic allocator may accidentally spill high-priority registers. On the other hand, when a spill occurs, the PBQP allocator repeatedly solves the remaining allocation problem until there are no more spills, preventing bad spills caused by evictions. Comparing Greedy-SO and *gs-basic* to isolate the performance impact of the fall-back register allocator with the same cost models and the same target functions, Greedy-SO provides 7% and 8% speedup on Intel and AMD CPU without any performance degradation (no red bar in Figure 9), while *gs-basic* struggles with performance degradation.

Based on the result of the sensitivity study, we used PBQP allocator as a fall-back register allocator for Greedy-SO and a threshold of 0.1 for the Greedy-SO register allocator.

6.3 Compilation Overhead

Greedy-SO introduces the overhead of recompilation only for the functions identified by both of the cost models as shown in Figure 3. The overhead of performing additional Greedy-SO register allocation path including IR function cloning and reverting logic is 1.7% (1.1% in AMD) of the total compilation time for LLVM CTMark benchmarks. Cloning function at machine function or MIR level or reverting only the register allocation result could further reduce recompilation overhead, but it is a non-trivial task [1, 5] and not in the scope of the paper.

7 Related Work

Linear scan register allocators [22] have been widely adopted as a cost-efficient alternative to the graph coloring-based register allocator [8]. [24] showed that the linear scan allocator can seamlessly combine with SSA forms [20] to outperform the coloring-based algorithms, and LLVM 3.0 further improved its linear scan allocator to use priority-based allocation [12] and enable global live interval splitting. Greedy-SO focuses on solving suboptimal decisions in these splitting heuristics.

[27] identified pathological cases caused by local interference not being considered during global-split in the greedy allocator. It examines if a region split may produce a local live interval that requires additional splits or spills due to interference and avoids splitting such candidates by boosting their potential spill cost. While it improves performance for its target test cases, adverse side effects were observed as shown in our evaluation. [7] recently reported that regional interference across multiple basic blocks should also be checked if they may introduce additional spilling, but the issue has not been addressed yet. Instead of tackling issues one by one at the heuristic design level, Greedy-SO systematically exploits spill cost modeling to avoid suboptimalities at a higher level.

There has been a recent effort to use hybrid or mixed register allocation schemes for efficiency and flexibility. [10] chooses between linear scan and graph coloring register allocators by comparing the spill costs of the two allocators to generate labels and train a rule induction model with them. Greedy-SO also uses spill cost tracking, but it builds a cost-guided optimization that integrates the two register allocators instead of choosing either. It also uses a high-level code pattern recognizer for more fine-grained hybrid register allocation. [16] dynamically chooses a register allocator for small code segments called “traces” using allocation policies based on live interval analysis, loop depth, block frequencies, etc. While Greedy-SO is different in reusing the existing global register allocators with minimal implementation overheads, adapting the allocation policies for Greedy-SO will be interesting future work. [15] suggested feedback-directed JIT compilation frameworks that formulate register spills as an ILP problem and make spill decisions using solutions from previous compilation. Greedy-SO currently focuses on improving the register allocator for AOT (Ahead-Of-Time) compilation with a single re-compilation.

8 Conclusion and Future Work

In this paper, we proposed Greedy-SO with the cost models for hybrid register allocation. In an effort to overcome the inherent limitations of heuristic fine-tuning, Greedy-SO provides a systematic way to detect suboptimal heuristic decisions and bypass them altogether and to do so only when performance benefit is expected. Our experiment showed that Greedy-SO could outperform the LLVM greedy allocator for target benchmarks without impacting non-target benchmarks, unlike prior work. Our future work includes extending our approach to other back-end code generation phases and introducing more cost-guided optimization passes, targeting other CPU and GPU platforms, and modeling learned predictors for the spill cost and code patterns to reduce the compilation overhead and improve the accuracy.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. 2019-0-01906, Artificial Intelligence Graduate School Program (POSTECH)) and the Super Computer Development Leading Program of the National Research Foundation of Korea (NRF) funded by the Korean government (Ministry of Science and ICT (MSIT)) (No. 2020M3H6A1084853).

References

1. Cloning machinefunctions, https://groups.google.com/g/llvm-dev/c/DOP6RSV8lQ4/m/C3Lfe6gJEwAJ
2. Intel(r) 64 and ia-32 architectures optimization reference manual, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
3. Llvm testing infrastructure, https://github.com/llvm/llvm-test-suite
4. Parallel bicubic interpolation, https://github.com/srijanmishra/parallel-bicubic-interpolation
5. Suggestions on register allocation by using reinforcement learning, https://groups.google.com/g/llvm-dev/c/V9ykDwqGeNw/m/-3SfJsuRAQAJ
6. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley (1986)
7. Brawn, J.: Strange regalloc behaviour: one more available register causes much worse allocation (Dec 2018), https://lists.llvm.org/pipermail/llvm-dev/2018-December/128299.html
8. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 428–455 (1994)
9. Callahan, D., Koblenz, B.: Register allocation via hierarchical graph coloring. ACM Sigplan Notices **26**(6), 192–203 (1991)
10. Cavazos, J., Moss, J.E.B., O'Boyle, M.F.: Hybrid optimizations: Which optimization algorithm to use? In: International Conference on Compiler Construction. pp. 124–138. Springer (2006)
11. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer languages **6**(1), 47–57 (1981)
12. Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. **12**(4), 501–536 (Oct 1990). https://doi.org/10.1145/88616.88621, https://doi.org/10.1145/88616.88621
13. Cohen, W.W.: Fast effective rule induction. In: Prieditis, A., Russell, S. (eds.) Machine Learning Proceedings 1995, pp. 115–123. Morgan Kaufmann, San Francisco (CA) (1995). https://doi.org/https://doi.org/10.1016/B978-1-55860-377-6.50023-2, https://www.sciencedirect.com/science/article/pii/B9781558603776500232
14. Cooper, K.D., Torczon, L.: Engineering a Compiler (Second Edition). Morgan Kaufmann, Boston, second edition edn. (2012). https://doi.org/https://doi.org/10.1016/B978-0-12-088478-0.00001-3, https://www.sciencedirect.com/science/article/pii/B9780120884780000013

15. Diouf, B., Cohen, A., Rastello, F., Cavazos, J.: Split register allocation: Linear complexity without the performance penalty. In: International Conference on High-Performance Embedded Architectures and Compilers. pp. 66–80. Springer (2010)
16. Eisl, J., Marr, S., Würthinger, T., Mössenböck, H.: Trace register allocation policies: Compile-time vs. performance trade-offs. In: Proceedings of the 14th International Conference on Managed Languages and Runtimes. pp. 92–104 (2017)
17. Evlogimenos, A.: Improvements to linear scan register allocation (2004), https://github.com/llvm/llvm-test-suite
18. Hames, L., Scholz, B.: Nearly optimal register allocation with pbqp. In: Joint Modular Languages Conference. pp. 346–361. Springer (2006)
19. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
20. Mössenböck, H., Pfeiffer, M.: Linear scan register allocation in the context of ssa form and register constraints. vol. 2304, pp. 229–246 (04 2002). https://doi.org/10.1007/3-540-45937-5_17
21. Olesen, J.S.: greedy register allocation (Sep 2011), http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html
22. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) **21**(5), 895–913 (1999)
23. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 12–27. POPL '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/73560.73562, https://doi.org/10.1145/73560.73562
24. Sarkar, V., Barik, R.: Extended linear scan: An alternate foundation for global register allocation. In: International Conference on Compiler Construction. pp. 141–155. Springer (2007)
25. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems. pp. 139–148 (2002)
26. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. pp. 132–141 (2005)
27. Yatsina, M.: Improving region split decisions (Apr 2018), https://llvm.org/devmtg/2018-04/slides/Yatsina-LLVM Greedy Register Allocator.pdf