Improving Region Analysis for Parallel Analysis of Programs

Buse Yilmaz BUSE.YILMAZ@OZU.EDU.TR
Baris Aktemur BARIS.AKTEMUR@OZYEGIN.EDU.TR
Computer Science, Özyeğin University

Abstract

A compiler performs several passes over a program during the semantic analysis and optimization phases before emitting executable code. The time required to do program analysis can be substantially reduced by performing it in parallel. Lee, Ryder and Fiuczynski's *Region Analysis* is a method that partitions the flow graph of a program to enable better load balancing of analysis tasks. In this study we show that Region Analysis' results can be improved by using simple heuristics. Numerous tests are done with different problem types and sizes, giving promising results. As a future work we intend to apply these heuristics on real-world large-scale applications and implement parallel program analyses.

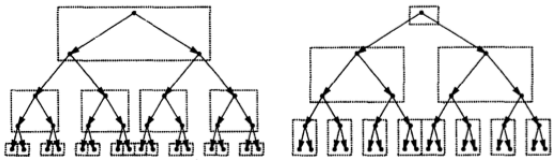
1. Introduction

Program analyses play an important role in the daily life of a software developer. For example, a compiler performs several analyses on a program to check if the program is compilable or if certain transformations are applicable; various optimizations that improve the efficiency of code have to use program analysis results; refactoring engines analyze programs to make sure that certain transformations are eligible. Several different analyses need to be performed by compilers and code transformation tools, such as data-flow, interval, shape, pointer, control-flow analyses. Furthermore, a transformation may invalidate some analyses by changing the program graph; therefore, an analysis may need to be recalculated several times during compilation. Thus, analyses may take substantial amount of time during compilation. To remedy this problem, program analyses can be performed in parallel (e.g. (Edvinsson et al., 2011)). Parallelizing an analysis has two sides: First, the algorithm must be made parallel. This requires working on the specific analysis problem and identifying opportunities for parallelization. Second, the input to the parallel algorithm, which is typically the program's flow graph, must be partitioned to be distributed to parallel tasks. A well-balanced partitioning is crucial in making a parallel analysis produce substantial speed-up.

Algorithm 1 Lee et al's method of expanding a region

```
for all node_k in region_j do
  for all c ∈ children(node_k) do
    if ∄ i . c ∈ region_i ∧ parents(c) ⊆ region_j ∧
    size constraint not violated then
      include c in region_j
    end if
  end for
end for
```

Region analysis (RA) (Lee et al., 1995) is a method that partitions flow graphs into disjoint, single-entry, multiple-exit subgraphs, called *regions*. RA depends on the following idea: Refining large graphs into smaller ones will offer a better degree of parallelism and load balancing, assuming equally likely computation costs among nodes. There are two constraints for forming a region. (1) Entry Constraint: a region has only one entry node which is its *head-node*. (2) Size Constraint: a region contains at most *S* nodes, where *S* is given as a parameter. Lee et al. (1995) provide two algorithms for region analysis. (1) forward algorithm, (2) bottom-up algorithm. Both are deterministic, greedy algorithms. The forward analysis is initialized by adding the root node of the graph to the first region. A new node is added to a region if all of its immediate predecessors are in the region and the size constraint is satisfied. This is how a region can grow until the size constraint is met. See Algorithm 1 for the pseudocode. When a region reaches the maximum size, a new region is created with a head node that is reachable from existing regions. The algorithm hence "moves" in forward direction and maximizes the number of nodes in a region before creating a new one. However, this greedy aproach may result in suboptimal partitionings in some situations; an example, taken from (Lee et al., 1995), is given in Figure 1. There are several potential region configurations of a graph, but the forward algorithm does not explore these possibilities. The bottom-up algorithm has similar drawbacks.



(a) (b)

Figure 1. Partitioning a graph into regions when the size constraint is 3 (taken from (Lee et al., 1995)). (a) The output of Lee et al's forward algorithm. (b) The optimal partitioning.

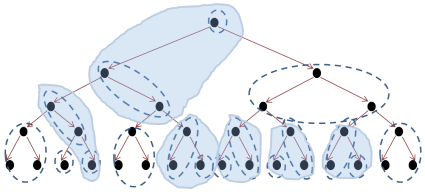


Figure 2. Region configuration using heuristic **H2** (blue dotted lines), and potential merges (blue transparent areas).

Contribution: Our work is based on the observation that Region Analysis can be considered a grouping problem such as the graph coloring problem or the knapsack problem. There are many state-of-the-art algorithms in the literature (Hertz et al., 2008; Galinier & Hao, 1999; Lü & Hao, 2010) targeting grouping problems. These algorithms are examples of meta-heuristics and hybrid approaches yielding better-quality solutions for sufficiently large graphs, compared to their counterparts with the greedy approach. Hence, it is expected to have a better partitioning of a flow graph when grouping techniques are applied to region analysis. Based on this observation, we propose an extension to Lee et al.'s algorithm that relaxes its strict greediness and determinism. We provide experimental results showing that partitionings with better load balancing can be achieved. In many cases, our extension is able to achieve optimal partitionings.

2. Our Approach

Our goal in this paper is to improve on Lee et al.'s forward algorithm's region partitioning by using heuristics from the grouping problem domain. The set of all valid region configurations (i.e. those that satisfy the entry and size constraints) of a flow graph form the search space. The optimal solution is the partitioning(s) with the least number of regions. We search for the optimal solution in the search space using a local search operator (LSO). The forward algorithm always starts from the root node and proceeds deterministically, maximally filling up a region before creating a new one. To make it possible that a variety of region partitions are created, we employ the following heuristics.

H1: Head nodes are picked deterministically as in the forward algorithm, but the size constraint S is not kept constant. Instead, a random size in $[1, S]$ is picked for each region created.

H2: Head nodes are picked randomly. The size constraint is constant, S.

H3: Head nodes are picked randomly. For each new region, size is randomly picked in range $[1, S]$.

Because we relax the size constraint independently on each region in **H1** and **H3**, we may produce a partition where there are several regions with small number of nodes. These regions have the potential to be merged so that the configuration's quality is improved. For this reason, we employ merging as our LSO. We first randomly pick a size S_R in range $[1, S)$. Then a region R with size S_R is chosen randomly. If there exists another region R' with size $S_{R'}$ such that $S_R + S_{R'} \leq S$, then R and R' are checked against possible violation of the entry constraint. If the constraint is satisfied, R' is merged into R. Note that not every merge attempt can succeed due to constraint violations. The ratio of successful attempts to the total number of attempts gives the *merge rate*. Merge rate can be used to adjust the number of merge attempts performed. Figure 2 shows the potential solution obtained for the graph given in Figure 1. Heuristic **H2** produces a configuration with 17 nodes (denoted with blue dotted lines). Merging (denoted by blue transparent areas) can reduce the number of regions to 11. Note that the forward algorithm produces a 21-node partitioning (Figure 1-a), and the optimal solution has 11 regions (Figure 1-b).

In the genetic algorithm terminology, ours is a population-based approach where an *individual* of a population is a partitioning of a flow graph into n regions. The number of regions, n, depends on the particular heuristic used. LSO is applied on each individual for *mergeCount* times to find an improved configuration. The general algorithm is given in Algorithm 2.

Algorithm 2 Population-based approach to region analysis using heuristics.

```
input: heuristic H
initialize the population
while iteration count not reached do
  for all individual_k in the population do
    construct a region configuration based on the heuristic H
    while mergeCount not reached do
      Apply LSO to individual_k
    end while
  end for
end while
```

3. Experimental Results

In this section, experimental results for forward algorithm and the heuristics along with the optimal number of regions are provided. We experimented with several graphs and obtained promising results.

In Table 1, we list the results for 6 sample program graphs taken from various papers. In these tests, our algorithm has been iterated for 50 times. Size constraint was 8. We used a population count of 20. For each individual, LSO is applied 30 times (i.e. *mergeCount* is 30). *mergeRate* indicates the ratio of successful merge attempts to the total number of attempts. For each input, we give the number of regions in the optimal solution and the number of regions constructed by the forward algorithm. For heuristics H1, H2 and H3, we give the average, the minimum and the maximum number of regions found. Note that minimum number indicates the best case. Except for the `bottom_up` graph, heuristics are able to achieve the optimal solutions.

4. Conclusion

In this study, a population-based local search algorithm for partitioning control flow graphs is proposed, with the aim of providing better load balancing for parallel analysis of programs. Along with three different heuristics for creating region configurations, a simple, yet effective local search operator is proposed. The experimental results show that the proposed approach is able to achieve better results than the forward algorithm of (Lee et al., 1995), and get very close to the optimal solutions. As future work we plan to benchmark bigger graphs, parallelize several analyses and finally implement our approach for the LLVM compiler targeting multi-core machines.

References

Edvinsson, M., Lundberg, J., & Löwe, W. (2011). Parallel points-to analysis for multi-core machines. *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (pp. 45–54). New York, NY, USA: ACM.

Galinier, P., & Hao, J. (1999). Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, *3*, 379–397.

Hertz, A., Plumettaz, M., & Zufferey, N. (2008). Variable space search for graph coloring. *Discrete Applied Mathematics*, *156*, 2551–2560.

Lee, Y.-F., Ryder, B., & Fiuczynski, M. (1995). Region analysis: a parallel elimination method for data flow analysis. *Software Engineering, IEEE Transactions on*, *21*, 913 –926.

Lee, Y. F., & Ryder, B. G. (1994). Effectively exploiting parallelism in data flow analysis. *The Journal of Supercomputing*, *8*, 233–262.

Lü, Z., & Hao, J.-K. (2010). A memetic algorithm for graph coloring. *European Journal of Operational Research*, *203*, 241 – 250.

Rountev, A., Kagan, S., & Marlowe, T. (2006). Interprocedural dataflow analysis in the presence of large libraries. In A. Mycroft and A. Zeller (Eds.), *Compiler construction*, vol. 3923 of *Lecture Notes in Computer Science*, 2–16. Springer Berlin / Heidelberg.

	Input graph					
Approach	(Lee et al., 1995)	5_level_tree	irreducible	(Rountev et al., 2006)	(Lee & Ryder, 1994)	4_level_tree
optimal	**2**	**5**	**1**	**5**	**3**	**2**
forward	3	10	3	5	4	8
H1 (mean)	5.30	10.22	3.21	5.79	3.81	4.40
H2 (mean)	5.92	8.15	3.20	6.27	3.95	3.67
H3 (mean)	4.20	9.01	3.18	5.50	3.93	5.00
H1 (min)	**3**	**5**	**3**	**5**	**3**	**2**
H2 (min)	**3**	**5**	**3**	**5**	**3**	**2**
H3 (min)	**3**	**5**	**3**	**5**	**3**	**2**
H1 (max)	7	17	5	8	7	8
H2 (max)	9	13	5	12	6	8
H3 (max)	7	17	5	8	7	8
H1 (mergeRate)	.1650	.4049	.0719	.4810	.1806	.2450
H2 (mergeRate)	.3272	.8450	.0750	.8986	.3300	.4360
H3 (mergeRate)	.1976	.4478	.0244	.3438	.1886	.2450

Table 1. Comparison of forward algorithm with heuristics.