A Region-Based Compilation Infrastructure[1]

Yang Liu Zhaoqing Zhang Ruliang Qiao
Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, PRC
{ly,zqzhang,qrl}@ict.ac.cn
Roy Dz-ching Ju
Microprocessor Research Labs, Intel Labs, Santa Clara, CA 95052, USA
roy.ju@intel.com

Abstract:

The traditional framework for back-end compilation is based on the scope of functions, which is a natural boundary to partition an entire program for compilation. However, the sizes and structures of functions may not be the best scope for program analyses and transformations when considering compilation resources (e.g. time and space), performance, and functionality. This problem is particularly pronounced when modern compiler optimizations resort to sophisticated and expensive algorithms to aim at high performance computing. Furthermore, it is often beneficial to give priority to optimize the more profitable portions of programs. Earlier works have proposed ways to allow some control on the size and structure of optimization scope. In this paper, we develop a new region-based compilation framework driven by the considerations of performance opportunities and compilation resources. In addition, we allow some optimization-directed attributes communicated from one optimization phase to another on a region basis to guide subsequent optimizations. This region-based framework has been implemented in the Open Research Compiler targeting Itanium® Processor Family (IPF). Experimental results from the SPEC2000Int programs show that this infrastructure provides an effective control on forming regions to meet the requirements of different optimizations. For example, the compilation time of instruction scheduling is significantly reduced by this region formation infrastructure while preserving or improving the overall performance. At the highest optimization level, the performance of eon program has a 15.6% improvement by employing this region-based infrastructure.

Keywords:

Region, Interval, Single-Entry-Multiple-Exit (SEME) Region, Multiple-Entry-Multiple-Exit (MEME) Region, and Compiler Optimization

1 Introduction

In order to exploit higher level of instruction level parallelism (ILP), modern compilers often resort to sophisticated and expensive program analyses and transformations, which typically consume substantial amounts of compilation time and memory usage. One way to keep the resource usage under control is to partition the traditional optimization scope of functions into smaller program regions, because these expensive algorithms typically have a complexity higher than linear with respect to the size of optimization scope. The notion of regions has been used in the past mostly restricted to particular optimization phases, such as code scheduling and register allocation. A region is informally defined as connected components in a control flow graph (CFG). Various regions have been proposed in previous work. Examples include trace [1], superblock [2], hyperblock [3], and region proposed by Hank et al [4][5].

To construct a flexible and general region-based compilation framework suited for different optimization phases, there are at least the following four aspects of a region should be considered.

a. *Region size*: The size of a region needs to be sufficiently large to obtain enough performance

[1] This project is supported by Intel Corporation and Nation Foundation of Natural Sciences 69933020.

opportunities but not too large to consume substantial compilation resources.

b. *Region shape*: The shape of a region also has a significant impact on the effectiveness of many optimizations. For example, a linear region provides no opportunity for if-conversion.

c. *Side entries to regions and tail duplication*: Many optimizations prefer to operate on single-entry-multiple-exit (SEME) regions, where side entries can be eliminated through tail duplication [2][3]. However, excessive code duplication greatly increases the code size, which may have an adverse effect on the efficiency of I-cache and hence program performance. Therefore, the amount of tail duplication must be controlled in order to avoid excessive code expansion.

d. *Region-specific characteristics*: Different parts of a program may have their specific requirements or characteristics to be observed during optimizations. A region provides a good boundary to annotate and observe these characteristics. For example, it may be desirable to specify a region to be processed by certain phases, but not by the others.

In this paper, a flexible and general region-based compilation infrastructure is proposed to address the requirements listed above. It partitions programs into regions as the optimization scope and has been implemented in an IPF Open Research Compiler (ORC). ORC is based on the open source Pro64 compiler from SGI and provides users a powerful and efficient framework with many new infrastructure features and IPF optimizations. Both its performance and compilation time are comparable to that of an IPF production compiler.

This infrastructure has the following important features:

a. Sizes and shapes of optimization scope can be controlled to meet the needs of different optimization phases. Code duplication ratios can be controlled during the process of SEME region formation in order to avoid excessive code expansion.

b. All of the analysis and optimization phases in the backend can operate under the region framework according to our design. This provides a uniform scope for optimization opportunities and compilation efficiency.

c. Regions with specific characteristics can be maintained and observed when processed across different phases. Maintaining and observing region-specific characteristics is a new requirement to our region-based compilation framework and has not been addressed in the previous work

d. The region framework is flexible, and it allows regions to be constructed, deleted, and reconstructed at different phases in the backend. If users choose to, the regions can also be constructed once and used throughout the rest phases with incremental updates.

We show that this framework is effective in reducing compilation time and improving performance through 12 SPEC2000Int programs on Itanium® machines. Compilation time is reduced significantly, especially when inter-procedural analysis (IPA) and function inlining (thereafter referred to as the peak mode) are enabled. For program **crafty**, compilation time of instruction scheduling with region formation is reduced by 63.9%. At the same time, the performance of these programs is improved greatly with region formation. Compared with the performance without region formation, the performance with region formation is improved by about 15.6% for **eon** and 7.8% for **crafty** with an average of 3.6% at the peak mode. It demonstrates that this infrastructure provides a flexible control on forming regions to meet the needs of different optimization phases.

The rest of this paper is organized as follows. In Section 2, the structures of regions are described in detail. The definitions of different region types and attributes are given. Section 3 discusses the region formation algorithms used to form different kinds of regions. Section 4 shows our experimental results. Section 5 discusses the related work. In Section 6, we summarize this paper and point out future work.

2 Definitions and Structure of Regions

Under our proposed region framework, the size, shape, duplication ratio and the characteristics of regions can be controlled, and users are able to partition the programs into compilation units with desired size and properties for different analysis and optimizations.

2.1 Structure of the Regional Control Flow Graph

Regions are composed of a set of connected nodes and edges. Each node in the region represents either a nested region or a basic block. A directed edge is an edge connecting two nodes and represents the control flow transfer from the source node to the target node. Every region has a local control flow graph called regional control flow graph. Every edge has the profiling information of its execution frequency and probability attached. The back edge in a loop region is also removed from its regional CFG but is recorded separately in the region.

Figure 1 shows an example of regional CFG. The nodes of the loop shown in **Figure 1(a)** form a region, and the regional CFG becomes the one in **Figure 1(b)**. The node R1 represents a nested region, which is formed for the loop of nodes {2,3}.

2.2 Structure of Regions

Relations among regions are organized hierarchically as a tree structure. This tree is called region tree. Root of the region tree is the outermost region, which corresponds to a whole function. Leaf nodes in the region tree represent the innermost regions.

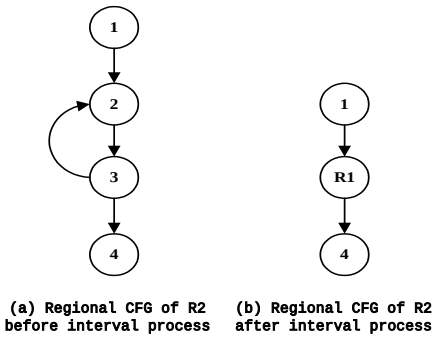

Figure 1 Regional CFG example

The nesting relationship between regions is represented as a parent-child relationship in the region tree. Here we also call a nested region a kid region. For example, region R1 in **Figure 1(b)** is a nested region of R2, i.e. a kid region of R2. The corresponding region tree is shown in **Figure 2**. R1 is R2's kid region, and R2 is R1's parent region.

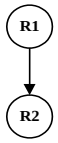

Figure 2 Region tree structure

2.3 Types of Region

In our framework, regions are classified into four types:

- Loop Region
- Improper Region
- Single-Entry-Single-Exit Region (SEME Region)
- Multiple-Entry-Multiple-Exit Region (MEME Region)

The four types of regions are described in detail below.

Loop Region Cycles in CFG become a structure boundary for regions. A loop region has a single entry, and the loop body is usually regarded frequently executed.

Improper Region Improper region contains irreducible control flow sub-graph. The cycle that contains the irreducible graph forms an improper region. Since irreducible graphs account for only a small portion of all programs and usually present limited optimization opportunities, they will be marked improper and treated specially by later optimization phases. For any region other than an improper region, its regional CFG is guaranteed to contain no cycles.

SEME Region SEME region is a region with single entry and multiple exits with possible control flow transfer within the region.

MEME Region MEME region has multiple entries and multiple exits also with possible control flow transfer within the region.

Unlike loop and improper regions, SEME and MEME regions are not formed due to any structure constraints, and instead they are formed usually by considering the following factors.

- **Exit Probability**

An SEME or MEME region can have multiple exits. The exit node with the max completion probability is the main exit node. The completion probability of an exit is defined as the possibility of the control flow going through this exit node while leaving the region. The completion probability of the main exit node is the main exit probability. Main exit probability is of importance to some optimizations and the shape of regions. For example, in the thread speculation in multi-threading parallelism, the benefit is largely affected by the main exit probability.

- **Duplication ratio**

The amount of duplicated codes e.g. from tail duplication, can be controlled by a duplication ratio in the process of forming SEME or MEME regions. Code duplication may increase optimization opportunities, but excessive code replication can have adverse effect, e.g. reducing the efficiency of I-cache. The trade-off can be controlled by this duplication ratio.

- **Region Size**

If a region is too large, the compilation time and space may become unacceptable. On the contrary, a small region may present too few optimization opportunities and reduce the optimization efficiency. The parameter of region size can be set as an upper bound while forming regions.

2.4 Attributes of Regions

Region attributes are proposed in order to keep the specific properties of regions consistent across different analysis and optimization phases. We currently have four kinds of attributes.

Persistent Boundary

Regions with this attribute must keep boundary persistent in different optimization phases. They can be divided into smaller regions, but their boundary restriction could not be violated, i.e., the entries and exits of a region cannot be changed. The purpose of this attribute is to protect the regions from being decomposed or combined with other regions. For example, an if-conversion phase may perform predication on the current region, and if this region is combined with other regions, it might make a later analysis of predicates generated from different regions less precise.

Rigid

Rigid region means the region cannot be decomposed any further. In different optimization phases, regions with this attribute must be kept as a whole. For example, the basic blocks in a region are if-converted to provide more instruction scheduling opportunities. If the region that has been if-converted is decomposed into smaller regions, some code motion opportunities will be lost. The rigid attribute will prevent the intermediate phases from decomposing the regions.

No Further Optimization

Once this attribute is set, no further optimization or decomposition can be performed to the region. Regions with this attribute must be kept intact in later optimization phases of compilation. This attribute is set for the reason that some regions have already been optimized and do not want any additional change. For example, a region already been software pipelined can be formed with this attribute to avoid being changed by later phases.

No Optimization Across Region Boundary

Any optimization across region boundary is inhibited. For example, removing two redundant computations from different regions (including nesting regions) with this attribute marked on either region is disallowed. Optimizations within the region boundary are allowed. This attribute ensures no changes in the live-in and live-out information of regions. For example, if a region is formed for multi-threading parallelism, its live-in and live-out sets may have been determined at the time of formation. We do not want subsequent phases to changes these sets, but we still want optimizations to be performed within the marked region boundary.

3 Formation of Regions

The region formation process can be divided into two steps. First, the root region is formed and intervals are identified and processed. In this framework, intervals include loops and irreducible strongly connected components, which are formed into loop regions and improper regions, respectively.

The next step is to decompose large regions into smaller SEME or MEME regions. Any region has a size greater than the max region size limitation will be decomposed into smaller regions to satisfy the constraint.

3.1 Formation of Intervals

Here the intervals include loops and irreducible strongly connected components. Loops are boundaries for many ILP optimizations and are identified first from the root region's regional CFG. Next, irreducible sub-graphs are detected and formed as improper regions.

The whole function is formed as a root region, and the global CFG is initially mapped to the root region's regional CFG. Cycles and back edges are computed using the algorithm described in [9]. A back edge is an edge with its source node dominated by its target node.

The regional CFG of the root region is traversed in a reverse preorder sequence to shrink loops from innermost to outermost. When the target node of a back edge is visited through a cycle, the back edge is identified and the respective loop scope is formed a loop region. All nodes in kid region shrink to one region node in parent region's regional CFG and control flow edges from these nodes are connected to the region node. The newly formed kid region is connected to the region tree.

Since each natural loop has reduced to a single region node in the regional CFG of their parent region, the strongly connected components (SCCs) left in the regional CFG must be due to irreducible control flow sub-graph. They are formed as improper regions.

3.2 Algorithm to Form MEME Regions

In our framework, because most optimization phases in backend are implemented based on SEME regions, MEME region formation is only an intermediate process of SEME region formation. But users could choose to form MEME regions only without SEME region formation for their needs. Our algorithm is detailed in **Figure 3**.

Like Hank's MEME region formation algorithm [4][5], the most frequently executed node is selected as a seed. Next, a path from the most frequent successor satisfying the following equation is extended from the seed. This process continues until the successor path can no longer be extended.

$$Succ(x,y) = \left(\frac{W(x \to y)}{W(x)} \geq T\right) \&\& \left(\frac{W(y)}{W(seed)} \geq T\_\right)$$

T and T_ are threshold values selected by a compiler. Similarly, *Pred(x,y)* is defined and a path of the most frequent predecessors is added next. The resulting seed path is further extended by selecting all most frequent successors satisfying *Succ(x,y)* from every node in the region. Selected nodes will then be added to the region and this process will continue until no node can satisfy *Succ(x,y)*.

In order to make the formed MEME regions more suitable for forming SEME regions with reasonable scope when exit probability and tail duplication ratio is required, we do two improvements on Hank's MEME region formation heuristics.

The first is that, though a similar selection of the seed path, a control on the length of seed paths is added, i.e., the seed path's length could not exceed **size/a**, where **size** is the max region size limitation and **a** is a threshold value greater than 1 selected by the compiler. It gives the seed path some chance to grow wider in order to avoid the shape of a thin trace.

As mentioned before, in our framework, the MEME region formation is a base for the SEME region formation to be described later. If the chosen MEME region has many side entries and we cannot afford a large code duplication ratio, the formed SEME regions may have a small size. Regions of small sizes usually significantly limit the opportunities to different optimizations. In order to reduce the scenario of many side entries, our approach makes a tradeoff between the shape of regions and execution frequency. When extending from the seed path, the nodes most frequently connected to the region are selected instead of always the most frequent successors (see function **Weight** in

Figure 3). Hence, our algorithms can usually select regions with larger sizes and more desirable shapes without heavily relying on tail duplication.

```
Find_MEME_Nodes(int size,regional_cfg cfg) {
   node_set R=∅;
   seed = the most frequent node in cfg;

   /* a is a threshold value defined by compiler */
   length = size / a;

   /* extend from seed to a seed path */
   x = seed;
   y = most frequent successor of x;
   while ((Size(R) < length) &&
          (y ∉ R) && Succ(x,y)){
     R = R∪{x};
     x = y;
     y = most frequent successor of x;
   }

   x = seed;
   y = most frequent predecessor of x;
   while ((Size(R) < length) &&
          (y ∉ R) && Pred(x,y)){
     R = R∪{x};
     x = y;
     y = most frequent predecessor of x;
   }

   /* expand seed path to a region */
   while (Size(R) < size) {
      max_weight = 0;
      for every node n ∈ R {
         for every succ of n and succ ∉ R {
            weight = Weight(succ);
            if weight > max_weight cand = succ
         }}
      R = R∪{cand};
   }
}

Weight(node x) {
   int weight=0;
   for every predecessor y of x {
      if (y∈R) {
         weight = weight + Edge_Freq(y→x);
      }}

   for every successor y of x {
      if (y∈R) {
         weight = weight+ Edge_Freq(x→y);
      }}
}
```

Figure 3 Algorithm of forming MEME regions

3.3 Algorithm to Form SEME regions

Instead of the more general MEME regions, many optimizations work effectively under SEME regions by retaining most optimization opportunities and without overly complex algorithms. Therefore, SEME region formation is of great importance. This paper proposed a new algorithm to form SEME regions by taking into account the code duplication ratio, exit probability, and max region size as outlined in Section 2.3. Our algorithm of selecting SEME regions is detailed in **Figure 4.**

Because exit probability and tail duplication ratio requirements, this SEME region formation algorithm is more complex. Algorithm begins from procedure Find_SEME_Region. First an MEME region is selected by calling procedure Find_MEME_Nodes with the algorithm described in Figure 3. For each node in the MEME region, we compute the scope if it is a main exit node by calling Compute_Scope_Base_On_Main_Exit.

Then, for each scope in MEME region, compute the main exit probability and compare it with the main exit probability requirement. For every scope which could satisfy the exit probability, compute the duplication ratio if tail duplication is allowed. If the code expansion exceeds the limit of duplication ratio, some dangling nodes are cut from the scope to reduce code expansion. This process will continue until the tail duplication ratio requirement is satisfied. Finally, a scope with max size is selected as the base to be tail duplicated. Then tail duplication is done to the scope and an SEME region is formed.

3.4 Discussions

In our framework, all regions are optimized one by one in the backend optimization phases. Most of these phases, e.g. instruction scheduling, have an $O(n^2)$ time complexity, where n is the number of instructions. By controlling the size of n in each region, the compilation time for optimizations based on SEME regions can be well constrained. By limiting the scope of optimizations to each region, for example instruction scheduling can only move instructions from one basic block to another within the same region. This could prevent instructions from being moved from basic blocks of very low execution frequency to those frequently executed ones so that optimizations can concentrate on regions with high execution frequency.

In our framework, we proposed a novel type of SEME regions controlled by tail duplication ratio and exit probability. SEME regions with single entries simplify the algorithm of many optimization phases, therefore reducing the development efforts and usually compilation time as well. With the flexible algorithm, users can set the parameters of tail duplication ratio and exit probability to adjust the shape of formed SEME regions to maximize the effectiveness of various optimizations.

```
float max_duplicate_ratio,min_exit_prob; int size;

Find_SEME_Nodes(regional_cfg cfg){
   node_set MEME_R = Find_MEME_Nodes(size,cfg);
   node_set R = ∅;

   for every node x ∈MEME_R {
      R' = Compute_Scope_Base_On_Main_Exit(x);
      exit_prob = Compute_Main_Exit_Prob(R',x);
      if (exit_prob > min_exit_prob) {
         dup_ratio = Compute_Duplicate_Ratio(R');
         if (dup_ratio > max_duplicate_ratio) {
            R'= Do_Selective_Cut(R',x);
            if Weight(R') > Weight(R) {
            R = R';
            }}}}
    return R;
}

Compute_Scope_Base_On_Main_Exit(node main_exit,
   node_set R){
   node_set del_nodes=∅;

   for every succeccor x of main_exit  {
      del_nodes = del_nodes∪{x}; }
   for every node x in depth first traverse order of R {
      if all predecessors of x are in del_nodes {
         del_nodes = del_nodes∪{x};
      }}
   R=R-del_nodes;
   return R;
}

Compute_Main_Exit_Prob(node_set R,node main_exit){
   /* Because tail duplication will change edge frequency of
      nodes to be duplicated, we must recompute them first */
   if there is any side entry to R {
      for  every node x∈R in toplogical traverse order of R {
         for every edge e of x{
            recompute frequency of e if tail duplicate is done;
         }}}

   total_freq = 0; exit_freq = 0;
   for every node x∈R {
      for every successor y of x  {
         if y ∉ R  {
            total_freq = total_freq + Edge_Freq(x→y);
         }}}
   for every successor y of main_exit {
      exit_freq = exit_freq + Edge_freq(main_exit→y); }
   return exit_freq/total_freq;
}

Do_Selective_Cut(node_set R,node main_exit){
   dangle_node_list = ∅;
   while (ratio > max_duplicate_ratio) {
      for every node x∈R  {
         if (all successors of x ∉ R && x!=main_exit ) {
            dangle_node_list = dangle_node_list∪{x};
         }}}
      if (dangle_node_list = = ∅) { return null; }
      d =least frequent node in dangle_node_list;
      R= R- {d};
      ratio = Compute_Duplicate_Ratio(R);
   }
   return R;
}

Compute_Duplicate_Ratio(node_set R){
   dup_nodes = all nodes should be duplicated in
                  order to eliminate side entries;

   return Size(dup_nodes)/Size(R);
}

Weight(node_set R){
   int  weight=0;
   for every node x∈R {
      weight = weight + number of ops in x*Freq(x);  }
   return weight;
}
```

Figure 4 Algorithm of forming SEME regions

4 Experiments

4.1 Implementation

Our region formation guided with profiling feedback is implemented as an early phase in the backend of ORC compiler. The majority of the phases in the backend operate under the region framework, and these include instruction scheduling, if-conversion, predicate analysis, software pipelining, loop unrolling, extended basic block optimizations, control flow optimizations, etc. Most of these optimizations perform effectively under SEME and loop regions. MEME regions is implicitly performed as part of forming SEME regions. Users of ORC can choose to construct MEME regions according to their requirements. Region attributes are created and checked to meet the need of the current implementation. Current settings of related parameters are: the maximum region size 20, the minimum exit probability 0, and no tail duplication.

Most of the optimization phases under the region framework make effective uses of the region scope to guide their optimizations. For example, if-conversion phase looks for patterns within SEME regions to identify candidate basic blocks for if-conversion. A subsequent predicate analysis phase analyzes the relations of predicates in individual SEME regions, which match the scope of the if-conversion without paying excessive compilation resources.

The instruction scheduler in ORC is extended based on the work proposed by D. Bernstein et al [13] to perform on SEME regions and performs partial-ready code motion [14]. The scheduler also makes use of predicate analysis to effectively schedule predicated code. So, forming regions with suitable sizes and shapes for this scheduler is very important. When region formation is turned off in the subsequent experiments, the scopes for the optimization phases are loops or functions as in a traditional compilation framework.

4.2 Results and Evaluation

The hardware platform for experiments is an Itanium® workstation with the processor speed of 733MHz and a 2M L3 cache. We use 12 Spec2000INT programs to evaluate the compilation time and performance. Because instruction

scheduling is the most time consuming phase in the ORC backend, it is chosen to evaluate the effect of region formation on compilation time.

Figure 5 and **Figure 6** compare the performance and compilation time of instruction scheduling at the peak optimization level, respectively, where the peak mode includes inter-procedural analysis (IPA) and function inlining in addition to O3 (all intra-procedural optimizations) and edge profiling feedback. (Due to space limitation, we are unable to show the comparison at different optimization levels.) The percentage of performance improved by region formation is computed using the following equation:

$$p = \frac{runtime_{without\_region} - runtime_{with\_region}}{runtime_{without\_region}} \times 100\%$$

Similarly, the equation used to compute the percentage of compilation time reduced by region formation is:

$$p_c = \frac{compiletime_{without\_region} - compiletime_{with\_region}}{compiletime_{without\_region}} \times 100\%$$

Figure 5 shows the performance comparison. Many programs have significant performance improvements when region formation is enabled. **Eon** has more than 15.6% performance improvement under region formation. **Crafty**, **perlbmk**, **parser** and **vortex** have their performance improved by 7.8%, 6.5%, 3.6% and 3%, respectively. Frequently executed basic blocks, which may spread across different functions before inlining, are placed into the same regions, and region-based optimizations can effectively optimize these highly profitable regions. Only **gzip** degrades a slight 1.4%. The average performance gain reaches 3.6% when region formation is enabled.

Figure 6 shows that forming regions has greatly reduced the compilation time of instruction scheduling. At the peak mode, region formation is important in reducing compilation time. For example**, crafty**'s compilation time under region formation is reduced by 63.9% compared to that without region formation. This can be understood that the functions in the Spec2000INT programs often have large scopes containing a large number of basic blocks, which make some complex algorithms, such as instruction scheduling, suffer a long compilation time. IPA and inlining make this situation worse. Region formation decomposes large functions into smaller units to limit the optimization scope for expensive algorithms and hence reduce compilation time. **Figure 7** shows the compilation time of backend reduced by region formation at the peak mode.

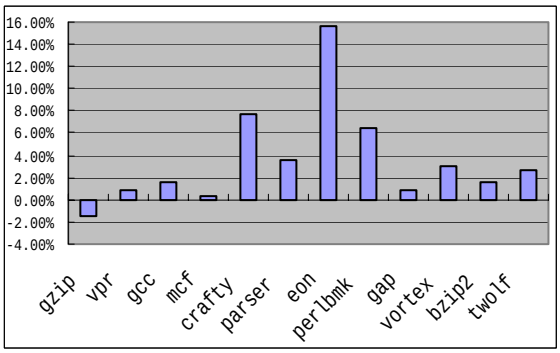

Figure 5 Spec2000Int Percentage of Performance Improved by Region Formation at Peak (the average improved by 3.6%)

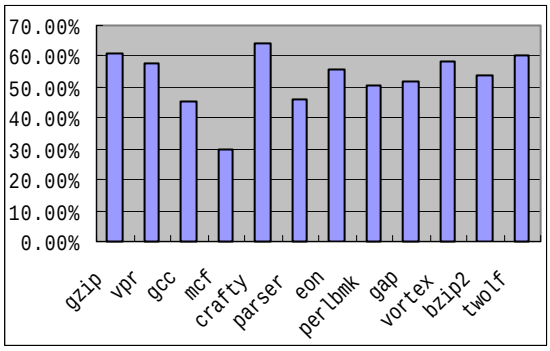

Figure 6 Spec2000Int Percentage of Instruction Scheduling Compilation Time Reduced by Region Formation at Peak (the average reduced by 52.8%)

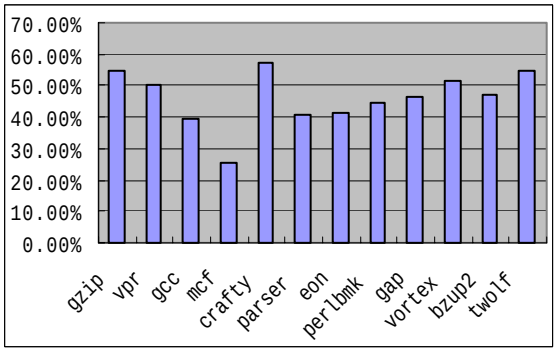

Figure 7 Spec2000Int Percentage of Backend Compilation Time Reduced by Region Formation at Peak (the average reduced by 46.1%)

From the implementation and data we can see that this infrastructure has a good control on forming regions to meet the requirements of different optimizations. Although decomposing functions into smaller units limits the optimization scope of many phases, it does not limit the effectiveness of these optimizations and instead contributes to good performance gains at the peak mode.

5 Related Work

In trace scheduling [1], Fisher proposed linear execution paths called traces as the scope for code scheduling. Fisher's traces may have multiple entries, and this leads to some complexities in ILP optimizations and code scheduling. In order to eliminate the side entries, superblock is proposed by Hwu et al in [2]. Superblocks are constructed based on execution profile information. Tail duplication is used in the superblock formation algorithm in order to eliminate side entries. Treegion [6], also targeting code scheduling, has the region shape of basic blocks connected as a tree structure. Treegion formation is not based on profile information but on the control flow analysis. No merge points exist within a treegion, which reduces the bookeeping in code scheduling. Region scheduling [7] uses an extended program dependence graph to partition programs into regions that have the same control conditions. In the extended program graph in [7], regions are organized in a hierarchical structure. Register allocation based on regions was also attempted in [4][8].

Comparison with Hank's Work

Hank used the notion of regions in various optimization phases to balance between performance opportunities and compilation resources [4][5]. Some optimizations based on regions were discussed, and an MEME region formation algorithm was presented. In their work, two aspects were compared between region-based compilation and function-based compilation: compilation time and code quality. In contrast, our region structure is different from Hank's and our work also provides additional contributions as follow.

I. In our framework, an SEME region formation algorithm with exit probability and tail duplication ratio control is proposed. This is totally different from the regions formed in Hank's framework, which concentrates on MEME region formation. SEME regions are the more effective scope for many optimizations.

II. We partition the whole program into regions, where every basic block is contained in a region. Hank's algorithm chooses only some basic blocks with high execution frequency to form regions. Furthermore, Hank's region structure is a flat one, without nested regions except for loops. In our framework, a region could nest in another region, and all regions are organized hierarchically.

III. In our framework, four region attributes are proposed. These attributes are checked and maintained across various optimization phases, and they carry important optimization-guiding information on a per-region basis.

IV. In our work, the performance from real machines with and without region formation is compared to demonstrate the effectiveness of a region-based infrastructure. Hank et al did not have measurements on execution performance.

6 Conclusions and Future Work

In this paper we proposed a flexible and general region formation infrastructure, and it has a number of features and advantages over prior work. With enhanced heuristics to form regions, our algorithm forms SEME regions of the shapes with a better balance between height and width, which are more suited to many advanced global optimizations and analyses. Exit probability and region size are used to control the size and shape of the regions formed. Our algorithm applies tail duplication judiciously to avoid excessive code size increase and adverse I-cache effect while forming effective regions.

This region framework has been developed in the backend of ORC to drive various optimization and analysis phases, and can be further extended to the whole compiler. Region attributes are proposed to annotate the properties of each region and communicate information from one phase to another on a per-region basis. The region structure can be

deleted and reconstructed at various phases. Experiments show that performance is often improved under region formation. Compilation time has also been reduced dramatically under region formation.

Although this infrastructure forms effective regions, the data dependence relationships across regions are not currently captured, which is useful to perform analysis across region boundary. We would like to form regions with a more sophisticated analysis of CFG and data dependence graph. In the meantime, more work need to be done to keep region attributes intact when processing across phases. We are also using this region-based infrastructure to build a compiler for multi-threaded architectures.

Reference

[1] J. Fisher, "Trace scheduling: a technique for global microcode compaction", IEEE Trans. on Computers, Vol. No. 7, pp. 478-490, 1981.

[2] W.W. Hwu,et al, "The Superblock: An effective way for VLIW and superblock compilation", The Journal of Supercomputing, vol. 7, pp. 229-248, January 1993.

[3] S. A. Mahlke, D. C. Liu, W. Y. Chen, R. E. Hank and R. A. Bringmann, "Effective compiler support for predicted execution using the hyperblock" In Proceedings of 25th International symposium of Microarchitecture, pp45-54,1992.

[4] R. E. Hank, "Region Based Compilation", Doctoral thesis, University of Illinois at Urbana Champaign,1996.

[5] R. E. Hank, W. W. Hwu and B. R. Rau, "Region Based Compilation:Introduction, Motivation and Initial Experience", International Journal of Parallel Programming, 25(2):113-146, Apr, 1997.

[6] W. A. Havanki, "Treegion scheduling for VLIW processors", MS Thesis, Dept.of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 1997.

[7] M. Gupta and M. L. Soffa, "Region Scheduling", IEEE Trans. on Software Engineering, vol. 16, pp. 421-431, April 1990.

[8] R. Gupta, M. L. Soffa and D. Ombres, "Efficient Register Allocation via Coloring Using Cluque Separators", ACM Trans. On Programming Languages and Systems, Vol. 16, pp370-386,May 1994.

[9] A. Aho, R. Sethi and J. Ullman, "Compilers: Principles, Techniques, and Tools", MA:Addison-Wesley, 1986.

[10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence" In Proceedings of the 10th ACM Symposium on Principles of Programming Languages, pp. 177-189, January 1983.

[11] Richard Johnson and Michael Schlansker. "Analysis techniques for predicated code" In Proceedings of the 29th Annual International Symposium on Microprogramming, pages 100-113, December 1996.

[12] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global predicate analysis and its application to register allocation" In Proceedings of the 29th International Symposium on Microarchitecture, pp. 114-125, December 1996.

[13] D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling" In Proceedings of 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture,1991.

[14] J. Bharadwaj, K. Menezes, and C. McKinsey. "Wavefront scheduling: Path based data representation and scheduling of subgraphs" In Proceedings of 32nd Ann. Int'l Symp. Microarchitecture (MICRO32), December, 1999.