2011 IEEE 17th International Conference on Parallel and Distributed Systems

Source Code Partitioning in Program Optimization

Murat Bolat
University of Delaware
Newark, DE, USA
murat@udel.edu

Kirk Kelsey
ET International, Inc.
Newark, DE, USA
kelsey@etinternational.com

Xiaoming Li
University of Delaware
Newark, DE, USA
xli@ece.udel.edu

Guang R. Gao
University of Delaware
Newark, DE, USA
ggao@capsl.udel.edu

***Abstract*—Program analysis and program optimization seek to improve program performance. There are optimization techniques which are applied to various scopes such as a source file, function or basic block. Inter-procedural program optimization techniques have the scope of source file and analyze the interaction and relationship between different program functions. The techniques analyze the entire translation unit (typically a source file) and optimize the whole translation unit globally instead of just optimizing inside a function.**

Analyzing and optimizing an entire translation unit increases compilation time drastically because many factors need to be considered during analysis and optimization. The translation unit size can be quite large, containing many functions. Another issue is that functions in different translation units can be more closely related to each other than to the functions within their translation unit.

The main goal of this research is grouping or partitioning of closely related program functions into the same translation unit. Our method profiles an application, determines relationship information between program functions and groups closely related functions together.

The source code partitioner method improves the processing time of inter-procedural optimization techniques by applying it to a subset of program functions. Partitioning of program functions by analyzing profiling output shows dramatic decrease in compilation time of programs. Our results show we can improve the compiling time in all tested real world benchmarks.

***Keywords*-compiler; optimization; partitioning; instrumentation; betweenness; agglomerative; divisive;**

I. Introduction

The goal of program optimization is to improve program performance. There are dozens of program optimization techniques in optimizing compilers. These optimization techniques are categorized in four types: peephole, local, inter-procedural and loop optimizations. The optimization techniques in the peephole, local and loop optimization categories are limited to function scope. Inter-procedural optimization techniques have a larger scope, extending over the entire translation unit [1].

Optimizing the interaction between program procedures can have significant impact on program performance, and today's complex compilers utilize this optimization category of inter-procedural optimization techniques. Complex program analysis and modifications are required for using inter-procedural program optimizations. The compile time suffers from this complex analysis. The standard optimization level (O3) of the GNU C compiler (GCC) [2] infrastructure contains inter-procedural program optimizations as well as local function level and basic block level program optimizations.

Because inter-procedural optimizations are applied to an entire translation unit, an increase in the size of the translation unit also increases the complexity of the application analysis. Achieving good performance impact requires tuning the number of functions included in a file in order to balance the impact of the optimizations with the cost of inter-procedural analysis.

Another consideration in inter-procedural optimization is the interaction between functions located in separate translation units. Optimizing this interaction by applying inter-procedural optimization techniques might have more significant impact than applying only within single translation units. In most libraries and tools, the source files are first compiled into object files while applying optimizations and linked together at a later stage. The interaction between functions in different translation units is rarely exploited.

The observation about a compiler's processing time in inter-procedural program optimizations raises an important question. How can we gain a significant performance increase while maintaining a reasonable compile time when using inter-procedural program optimizations?

This paper proposes a source code partitioner which divides the source code and creates reasonable translation unit sizes. The final partitioned translation units can be optimized more efficiently using the inter-procedural optimizations. The partitioning of the translation units is not done arbitrarily, rather the closely related functions are included into same translation unit. This function-to-function relationship is determined by application profiling.

The remainder of the paper is laid out as follows. Section II gives high level information about our source code partitioner, Section III talks about the source code profiler, Section IV deals with partitioning algorithms, Section V shows the application of the partitioner onto real world benchmarks and Section VI concludes the paper.

This work is funded in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650–09–C–7915.

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.125

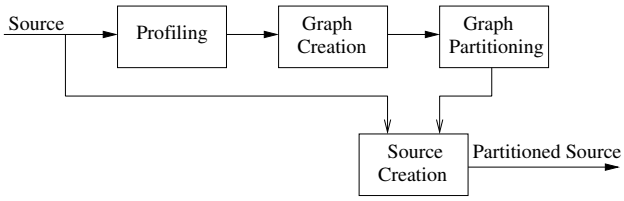


Figure 1. The source code partitioner.

II. Source Code Partitioner

This section has a high level description of the source code partitioner. The goal in our program partitioning is efficient program compilation while having good performance.

The major contributions in our source code partitioning approach are:

- Profile based partitioning of program files.
- Improved program compile time.
- Minimal increase in program execution time.

Our program partitioning approach implements two partitioning algorithms and creates new translation units as a result of partitioning. The partitioning also improves program compilation time. Only the partitioned source files are compiled with the highest compilation flag. The remaining source files are not optimized during compilation.

Our source code partitioner method has four phases:

1) Profile the application (*P*rofiling).
2) Create an undirected, weighted graph from profiling data (*G*raph Creation).
3) Partition the graph heuristically (*G*raph Partitioning).
4) Create translation units for each graph cluster (*S*ource Creation).

The flow graph of the source code partitioner is shown in Figure 1.

The first phase in our source code partitioner determines the interaction between program functions. The application is partitioned based on information about function interactions gathered from program profiling. We determine the relationship between program functions based on the source code profiler *suggestions for locality optimizations (SLO)* [3] described in Section III.

The profiling output is used to determine closely related program functions. The starting point is to create an undirected and weighted graph, which is later partitioned. The partitioning algorithms we have studied are the *community structure finding algorithm* and *scalable information bottleneck algorithm*. Section IV describes both algorithms in more detail.

Each partitioning result creates new source files (translation units). Functions in each graph cluster are moved into the corresponding source file. The remaining functions which are not included into the graph will be present in the original translation units. The created source files are later compiled into the binary and run. The compilation time and the run time are measured for the original program version and the partitioned version. Section V applies the source code partitioner onto real benchmarks and compares compile time and run time of the original version and partitioned version.

III. Source Code Profiler

The source code profiler analyzes the application in order to determine the relationship between the program functions. The current implementation uses the source code profiler *suggestions for locality optimizations* (*SLO*) [3]. This profiler is a modified GNU C compiler (GCC) [2] which includes profiling code while compiling the translation units. The SLO profiler is picked in order to optimize functions which access same memory locations.

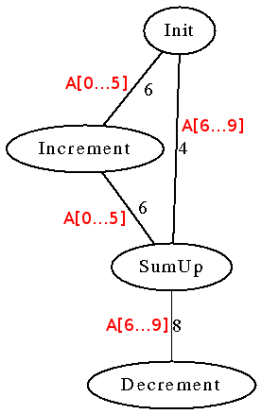
The SLO compiler also creates additional information files about each translation unit besides including profiling code. The additional files contain information about memory accesses. After program execution, the profiling database contains the memory reuse distances and how many times they occur. We can determine the distance and the occurrence number for each memory access pair. Using the profiling database and the additional information files, the memory access locations in translation units can be determined.

In our source code profiler, the memory reuse distances are not considered. Only the number of data reuses is used in creating the weighted, undirected graph. In the graph each node represents a function participating in a data reuse. The edge between two nodes represents the sum number of data reuses between that pair of functions. Processing profiling database creates a weighted and undirected graph. The functions participating in any data reuse in that particular run are shown as graph nodes. Those functions also participate in program partitioning. The remaining functions stay in the original translation units. The profiler determines memory reuses at the instruction level, where memory reuse among instructions are shown. Currently, we only look at function level and do not go into basic block or instruction level.

Figure 2(a) shows an example program and Figure 2(b) its corresponding undirected memory reuse graph. The profiler determines memory reuses among memory instructions. For example, the instruction in loop body of the function `Init` has six memory reuses with the instruction in loop body of the function `Increment`. The function `Init` stores values onto six array elements and the function `Increment` loads six of the stored values. There are two reuse cases between functions `SumUp` and `Decrement`. In the first case the function `SumUp` accesses the memory locations which the function `Decrement` later decrements. The second case is the opposite direction, where function `Decrement` decrements and function `SumUp` accesses. Both reuses are added together in the graph.

```
void Init (int *A, int N)
{
  for (int i=0; i<N; i++)
    A[i] = 0;
}
void Increment (int *A, int N)
{
  for (int i=0; i<=N/2; i++)
    A[i]++;
}
void Decrement (int *A, int N)
{
  for (int i=N/2+1; i<N; i++)
    A[i]--;
}
int SumUp (int *A, int N)
{
  int i, Sum = 0;
  for (i=0; i<N; i++)
    Sum = Sum + A[i];
  return Sum;
}
int main (void)
{
  int A[10], Sum;
  Init(A,10);
  Increment(A,10);
  Sum = SumUp(A,10);
  Decrement(A,10);
  Sum += SumUp(A,10);
  return 0;
}
```

(a) C Program



(b) Memory reuse graph

Figure 2. Example C program to demonstrate profiling tool and its corresponding memory reuse graph. The array locations participating in data reuses are shown in red.

The graph resulting from processing the profiling database is the starting point for the implemented partitioning algorithms.

IV. Partitioning Algorithms

The memory reuse graph is divided using graph partitioning algorithms. We have implemented the partitioning based on the *community structure finding (Community) algorithm* and the *scalable information bottleneck (LIMBO) algorithm*.

The partitioning algorithms use the edge weights in the graph and find a partitioning where internal edges have higher connection values than the external edges. The edges represent the sum number of data reuses among connecting node (function) pairs. The outcome will be the partitioning of the program. Sections IV-A and IV-B talk about the work flow of the partitioning algorithms. Section IV-C briefly describes the extraction of source code onto separate source files.

A. Community Structure Finding Algorithm

The *community structure finding (Community) algorithm* identifies communities, or clusters, which are closely related [4]. The paper proposes the algorithm in order to find community structures in large networks such as biological networks or the world wide web. We will use this algorithm in partitioning program functions into different translation units. The community structures are clusters with closely related parts in a graph. We want to see in our research how such a general clustering algorithm behaves in clustering with the goal of optimization. Wide usage of this algorithm affected our choice.

The algorithm uses a metric called edge betweenness. Edge betweenness calculation determines the popularity of an edge in routes between all node pairs. Although the work in [4] proposed the algorithm for an undirected, unweighted graph, we generalized it for a weighted graph.

The algorithm is a divisive algorithm that starts with the largest cluster and divides the cluster. The algorithm proceeds iteratively, and halts when the number of functions in each cluster is below a maximum value provided by the user. In each iteration, a new cluster is selected.

The algorithm has the following flow:

1) Calculate betweenness scores for all edges in the graph.
2) Find the edge with the highest score and remove it from the graph.
3) Recalculate betweenness for all remaining edges.
4) Repeat from step 2.

The paper [4] shows two different ways to calculate edge betweenness values which are shortest paths betweenness and random walks betweenness. Both ways result onto different partitioning. The source code partitioner contains shortest paths betweenness in order to calculate the betweenness scores.

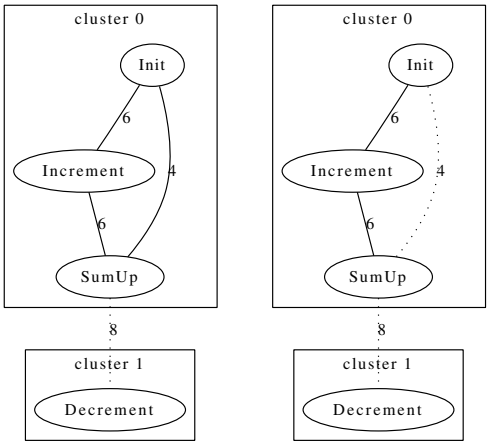
In shortest paths betweenness all-pairs shortest paths are calculated and the betweenness number is the number of times an edge is taken in shortest paths. The edge with the highest betweenness score is removed and the betweenness values are recalculated. The paper claims that the edge which is taken the maximum number of times is the edge connecting communities or clusters. The paper uses breadth-first search to determine a shortest path because the graphs in that paper are unweighted and undirected graphs. In the source code partitioner the Dijkstras algorithm [5] is implemented in order to calculate shortest paths.

The algorithm stops when there are no divisible clusters available. A cluster is considered non-divisible if the number of nodes (functions) are less or equal to the given function number parameter. The first drop from the function limit number is the stopping criteria for that cluster. The algorithm stops when all clusters reach the stopping criteria.

The processing of the Community algorithm is shown using our example memory reuse graph in Figure 2(b) of the example C program in Figure 2(a). The function number parameter value for this partitioning is two.

The shortest paths in the first iteration is shown below. The shortest path between nodes x and y is denoted as *SP(x,y)*. An edge between nodes x and y is denoted as *E(x,y)*. For each shortest path, the taken edges are shown.

- SP(`Init`,`Increment`): E(`Init`,`Increment`)
- SP(`Init`,`SumUp`): E(`Init`,`SumUp`)
- SP(`Init`,`Decrement`): E(`Init`,`SumUp`) → E(`SumUp`,`Decrement`)
- SP(`Increment`,`SumUp`): E(`Increment`,`SumUp`)



(a) After first interval (b) After second interval

Figure 3. The intermediate partitionings in Community algorithm.

- SP(`Increment`,`Decrement`): E(`Increment`,`SumUp`) → E(`SumUp`,`Decrement`)
- SP(`SumUp`,`Decrement`): E(`SumUp`,`Decrement`)

The edge betweenness value is the sum number of times an edge is taken in shortest paths. The edge *E(`SumUp`,`Decrement`)* is taken three times and has a maximum edge betweenness value of three. That edge is removed after this interval, and the partitioning in Figure 3(a) is created.

The *cluster 1* in Figure 3(a) is not considered in next interval because it has less or equal nodes than the function parameter value of two. The *cluster 0* will be further divided. The shortest path calculations at the second interval are:

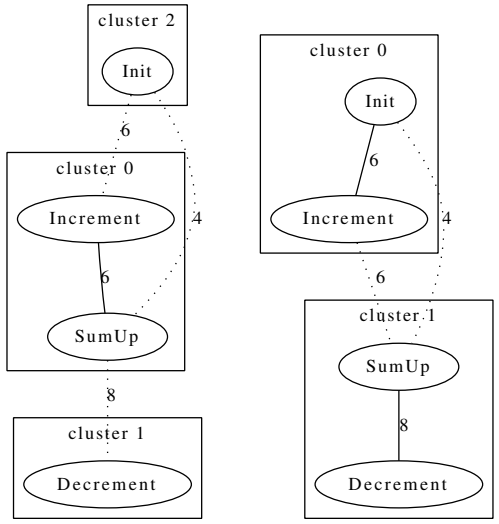
- SP(`Init`,`Increment`): E(`Init`,`Increment`)
- SP(`Init`,`SumUp`): E(`Init`,`SumUp`)
- SP(`Increment`,`SumUp`): E(`Increment`,`SumUp`)

All three edges in the *cluster 0* are taken one time and have an edge betweenness value of one. In this case the edge with a minimum edge number is chosen and removed, which is *E(`Init`,`SumUp`)*. After this interval, the number of nodes in the *cluster 0* does not change. The Figure 3(b) has the configuration after this interval. The shortest path calculations in interval three after that edge is removed are:

- SP(`Init`,`Increment`): E(`Init`,`Increment`)
- SP(`Init`,`SumUp`): E(`Init`,`Increment`) → E(`Increment`,`SumUp`)
- SP(`Increment`,`SumUp`): E(`Increment`,`SumUp`)

At this iteration both edges have betweenness value of two. In this case the edge to remove is chosen randomly, which is in our case the edge *E(`Init`,`Increment`)*.

The partitioning after this interval is shown in Figure 4(a). This is also the final partitioning because all clusters are less or equal the function number parameter of two.



(a) Community (b) LIMBO

Figure 4. The final partitioning in both algorithms.

B. Scalable Information Bottleneck (LIMBO) Algorithm

In contrast to the algorithm in the previous section, the *scalable information bottleneck (LIMBO) algorithm* is an agglomerative algorithm. The algorithm starts with single node clusters and combines clusters together. The algorithm, which uses ideas from information theory, is described in paper [6]. The paper applies the algorithm to decompose large software systems. The functions which are related to each other are included into the same cluster for better understanding the software systems hierarchy.

The paper proposes the algorithm for an unweighted graph which is generalized to a weighted graph in the source code partitioner. Another difference in the source code partitioner is that it only combines clusters which are connected with an edge to each other; in contrast, the paper allows combining any clusters together. The stopping criteria for combining two clusters is the function number parameter. The translation unit with that number is not considered in the next combination. The algorithm stops when no further combining for all clusters possible.

The algorithm uses an adjacency matrix **M**, where m_{ij} is the node weight between nodes i and j. The weight is zero if there is no edge between nodes. At each combination of nodes the row number of the matrix reduces, but the column number stays the same. The row headers are called as the artifacts **A** of the graph. **C** is the set of clusters, which has the size number of nodes/artifacts, initially. The column headers are the feature set **B** of the clusters, which has the size number of nodes. At each iteration, two rows are combined together and the element values are updated.

In order to apply ideas from information theory, the paper denotes **A** and **B** as two random variables, which can have

values of the sets. The relationship of those random variables leads to the combining clusters.

Initially, each artifact $a_1, a_2, \ldots, a_n \in A$ has a probability $p(a_i) = 1/n$, where n is the number of nodes in the graph. After calculating entropy and conditional entropy, the mutual information of both random variables is calculated.

The mutual information $I(A;B)$ for the random variables **A** and **B** is $I(A;B) = H(B) - H(B/A)$, where $H(B) = -\sum_{a_i \in A} p(a_i) \log_2(p(a_i))$ and $H(B/A) = -\sum_{a_i \in A} p(a_i) \sum_{b_j \in B} p(b_j/a_i) \log_2(p(b_j/a_i))$

The mutual information is the amount of dependency of one random variable from the other one. In our case the mutual information is the information about artifacts we can observe from the feature set. The goal is to create a clustering with high mutual information. The initial single clustering has highest mutual information and the combining of all nodes together has lowest mutual information. At each combination of rows the features set of both rows are averaged together, so the original information content is combined and leads to some loss of information.

The adjacency matrix **M** is normalized in order to get the value $p(b_j/a_i) = \frac{M[a_i,b_j]}{\sum_{b \in B} M[a_i,b]}$. The normalization results to the sum of one at each row.

Another set **C** is the clustering of the nodes, where **C** equals **A**, initially. The elements $c_1, c_2, \ldots c_n \in C$ equal the elements of **A**. As mentioned before, the goal is to have a high mutual information, so we need to combine two clusters which have a low information loss $\delta I(c_x, c_y)$. In order to calculate the information loss, the mutual information before combining is calculated and the mutual information after combining is calculated. The difference of the mutual information is the information loss. The clusters with lowest information loss are combined. For example, two nodes have the same number of edges, which connect to the same neighbors with same weight. Combining those two will have zero information loss because the resulting feature set has same values. This phase repeats iteratively until maximum number of nodes are reached in all clusters. That means further combining means clusters with size greater than the parameter maximum node number.

The formula for information loss of clusters c_x and c_y is $\delta I(c_x, c_y) = p(c_{xy}) \cdot D_{JS}[p(B/c_x), p(B/c_y)]$, where $p(c_{xy}) = p(c_x) + p(c_y)$ is combined probability, $p(B/c_{xy}) = \frac{p(c_x)}{p(c_{xy})} p(B/c_x) + \frac{p(c_y)}{p(c_{xy})} p(B/c_y)$ are combined features and $D_{JS}[p(B/c_x), p(B/c_y)]$ is Jensen-Shannon (JS) divergence which measures similarity between two probability distributions.

The information loss formula is used at each iteration between all clusters. The cluster pairs with the lowest information loss are combined together. The combined cluster has combined probability and combined features. This processing repeats until no combination possible. In our application, the artifacts are combined onto clusters until the function number parameter value is reached, so we don't want to have bigger cluster sizes than the function number parameter. In the original proposed algorithm, it is possible to combine nodes which are not neighbors, but in our source code profiler we do not combine non-neighboring nodes. These nodes have the same relationship with other functions in the application but do not interact with each other directly and will not have any benefit in inter-procedural optimizations.

	Init	Increment	SumUp	Decrement
Init	0→**0**	6→**0.60**	4→**0.40**	0→**0**
Increment	6→**0.50**	0→**0**	6→**0.50**	0→**0**
SumUp	4→**0.22**	6→**0.33**	0→**0**	8→**0.44**
Decrement	0→**0**	0→**0**	8→**1.00**	0→**0**

Table I. Adjacency matrix for the graph in Figure 2(b). Normalized values are in bold.

	Init	Increment	SumUp	Decrement
Init		**0.2770**	0.2806	
Increment	**0.2770**		0.3392	
SumUp	0.2806	0.3392		0.5000
Decrement			0.5000	

Table II. Information loss between rows (clusters) in table IV-B

	Init	Increment	SumUp	Decrement
Cluster 0	0.2500	0.3000	0.4500	0
SumUp	0.2222	0.3333	0	0.4444
Decrement	0	0	1.0000	0

Table III. New normalized adjacency matrix after combining nodes Init and Increment to cluster 0

	SumUp	Decrement
SumUp		**0.5000**
Decrement	**0.5000**	

Table IV. Information loss between rows (clusters) in table IV-B. The cluster 0 is not considered in this calculation.

As in Section IV-A we show the processing of this algorithm for the example graph in Figure 2(b). Creating the adjacency matrix **M** is the first step of the algorithm (Table IV-B). The matrix is normalized and each row sums up to one. The normalized matrix values are in bold. This matrix is used for calculating information losses in Table IV-B. The original algorithm considers all clusters in calculating information losses, but the modified version in this paper calculates only information losses between clusters with connecting edges. The information loss in Table IV-B shows combining nodes `Init` and `Increment` will lead to minimum information loss, so these nodes are combined into one cluster. In this example, function number parameter is two, and the new cluster has the size two, so it will not be considered in next combination process. This combination leads to the normalized matrix in Table IV-B. The information loss of this new normalized matrix is also calculated and the clusters to combine are determined (Table IV-B). There are only two single clusters `SumUp` and

`Decrement` left and they are combined together.

The final partitioning of the example data reuse graph in Figure 2(b) is shown in Figure 4(b). The final partitioning contains two clusters with two nodes. The *cluster 0* contains the functions `Init` and `Increment` and the *cluster 1* contains the functions `SumUp` and `Decrement`.

As can be seen in our example graph, both algorithms lead to different partitioning of the original data reuse graph for function number parameter of two.

C. Source Code Extraction

The partitioning algorithms create clusters, which group a subset of function (nodes) together. After program partitioning, the clusters are included in their own source files (translation units). This is the last phase in our source code partitioner. The functions in each cluster are extracted and included in their corresponding translation unit. The remaining functions still stay in their original translation unit. In each translation unit the required function prototypes, type (enum, struct and typedef) definitions and global variable declarations are included in the translation units.

For our example program in Figure 2(a), all functions but function `main` participate in partitioning. The `main` function stays in the original source file, and the other functions are moved to their corresponding source files. The outcome of the Community algorithm in Figure 4(a) indicates three additional source files wih corresponding functions. The partitioning of the LIMBO algorithm creates two additional source files for each cluster in Figure 4(b).

The function number parameter given by the user also has an effect on the partitioning outcome. In our example program, the function number parameter was two for both algorithms. Both algorithms determined different partitioning for that parameter value. For the case, we modify the parameter to three, both algorithms will determine same partitioning of the graph. The partitioning will result in the same clustering as the one in Figure 3(b).

V. Experimental Results

The impact of applying inter-procedural program optimizations on a partitioned subset of application functions is also measured on real world benchmarks. The source code partitioner is tested on six benchmarks. Those benchmarks are *401.bzip2*, *429.mcf*, *433.milc*, *458.sjeng*, *464.h264ref* and *470.lbm* of SPEC CPU2006 benchmark suite [7].

The compile and run time of the benchmarks are measured for the original, unpartitioned version, and for the partitioned version after the source code partitioner has been applied. The benchmarks are compiled and run for two cases:

1) Unpartitioned translation units compiled with standard optimizations (*Original Version*).
2) Partitioned translation units compiled with standard optimizations and remaining translation units without optimization (*Partitioned Version*).

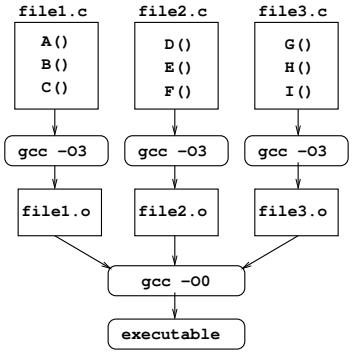


Figure 5. Compilation of an application (*Original Version*).

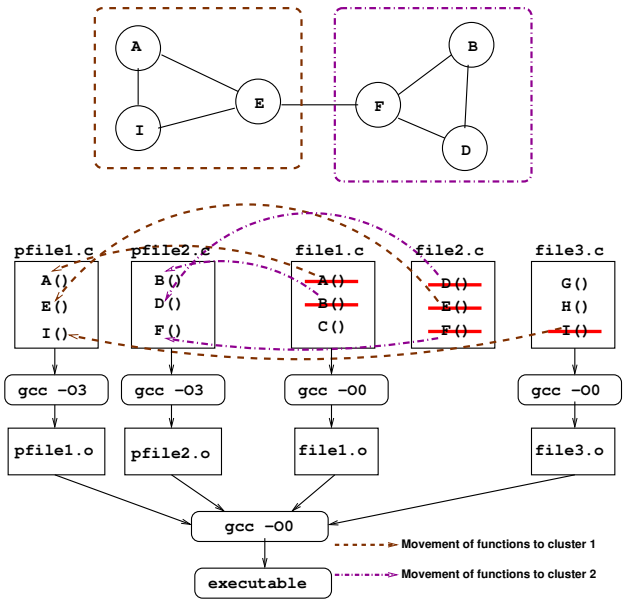


Figure 6. Partitioning of an application and compilation after partitioning (*Partitioned Version*)

The figure 5 shows the compilation of the original version of an application with three files (`file1.c, file2.c, file3.c`) and nine functions (`A, B, C, D, E, F, G, H, I`). The files are compiled with the GCC compiler version 4.4.3 [2] using the highest standard compilation flag `O3`. This flag contains inter-procedural program optimization techniques as well as local function level and basic block level program optimization techniques. The resulting object files (`file1.o, file2.o, file3.o`) are linked together and compiled to executable without applying optimizations which is the standard flag `O0`. After creating the undirected reuse graph, the application functions are partitioned as in the top part of the figure 6. The bottom part of the figure shows the modified compilation for the partitioned version. The functions in each cluster are moved from their original translation units to their corresponding cluster translation units (`pfile1.c,`

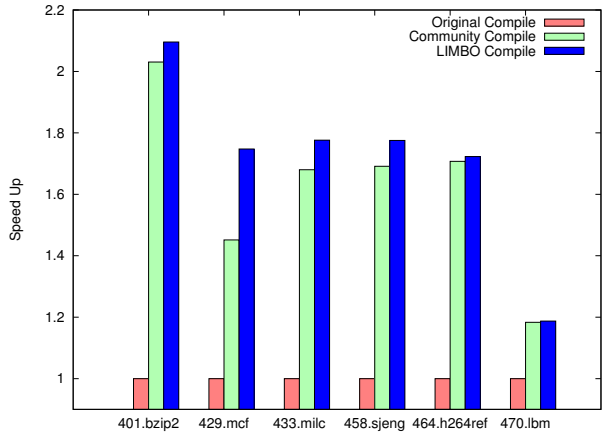


Figure 7. Compile time speed up for the five benchmarks. Partitioning time is also included in the compile time.

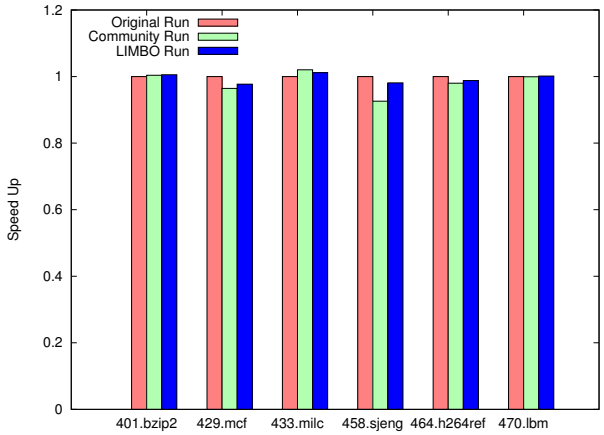


Figure 8. Run time speed up for the five benchmarks.

`pfile2.c`) and removed from their original translation units. The file `file1.c` contains only function `C`, and file `file3.c` contains the functions `G` and `H`. The cluster translation units (`pfile1.c, pfile2.c`) are compiled with the standard compilation flag and remaining translation units without applying any optimizations. The file `file2.c` is not compiled because all the functions are removed from it. The resulting object codes are linked and compiled to executable similar to the original version. The partitioned version has four files, two of them compiled with standard flag and two of them without optimizations.

The compile time and run time speed up can be seen in Figures 7 and 8. The speed up is shown for six benchmarks and for both algorithms. The compile time and run time speed up of the original benchmark is also shown. The compile time also includes the partitioning time. All values are normalized to the original benchmark values, which leads to speed up of one for the original case. The Community algorithm of Section IV-A is indicated as Community. The LIMBO algorithm of Section IV-B is indicated as LIMBO. The benchmarks are run with their training input data. The table V shows summary of the clustering for the six benchmarks. The summary is shown for six benchmarks and for both algorithms. The column *Partitioning* indicates the fraction of partitioning time in total compilation of the partitioned version. The last two columns show how many clusters with the given corresponding number of functions exists. For example for the Community algorithm in benchmark *401.bzip2*, there are two clusters with six functions/nodes.

The compilation speed up for the benchmark *401.bzip2* is 100% for the Community algorithm and 110% for the LIMBO algorithm. There is no run time slow down for both partitionings.

The benchmark *429.mcf* has a compilation speed up of 42% for the Community and 75% for the LIMBO algorithm. The run time slow down is 3% for the Community algorithms and 2% for the LIMBO algorithm.

There is speed up in compilation and run time for the benchmark *433.milc*. The compilation speed up is 47% for the Community algorithm and 76% for the LIMBO algorithm. The runtime improves 2% for the Community and 1% for the LIMBO. There are 68 files in this benchmark, with only functions from 48 files participate in partitioning. The remaining 20 files are compiled without applying optimization techniques, which lead to total compilation speed up. Grouping of the closely related functions from the 47 files reduces analysis time of the compiler and leads to compilation time reduction. The run time performance of the partitioned versions for both algorithms shows intelligent grouping of the program functions for big benchmarks improves the compilation time while having reasonable run time performance.

The benchmark *464.h264ref* has a compilation speed up of 29% for the Community algorithm and 71% for the LIMBO algorithm. The Community algorithm has 2% and the LIMBO algorithm 1% run time slow down. The compilation speed up is 18% for both algorithms in benchmark *470.lbm* and no run time slow down.

Our partitioning shows the Community algorithm creates one or two clusters with many nodes (dominating clusters) and many single node clusters. Applying inter-procedural optimizations on only those dominating clusters will further reduce compilation time while having similar run time performance. The functions in dominating clusters have a high degree of interaction with each other but the single node clusters create many translation units. Further improvement could be found by combining the single node clusters.

The LIMBO algorithm creates clusters with a more evenly distributed number of nodes. The overall partitioning leads to a reasonable number of translation unit sizes and number of translation units.

The results show smaller translation unit sizes with closely

Benchmark	Algorithm	Partitioning	# Clusters	# Functions
401.bzip2	Community	1.09%	2	6
			3	3
			9	1
	LIMBO	0.14%	2	6
			2	5
			3	3
			2	1
429.mcf	Community	2.34%	1	3
			2	2
			12	1
	LIMBO	0.32%	2	4
			3	3
			1	2
433.milc	Community	12.45%	1	12
			2	6
			1	5
			2	4
			1	3
			4	2
			19	1
	LIMBO	0.64%	1	13
			1	12
			2	11
			1	6
			1	5
			1	3
			3	2
458.sjeng	Community	11.34%	1	8
			1	6
			1	3
			5	2
			20	1
	LIMBO	0.61%	2	8
			1	7
			1	6
			1	5
			1	4
			1	3
			3	2
464.h264ref	Community	24.26%	1	21
			1	8
			1	7
			1	6
			3	4
			2	3
			7	2
			44	1
	LIMBO	0.72%	2	23
			1	21
			1	19
			1	16
			1	10
			1	4
			1	2
470.lbm	Community	0.15%	1	2
			4	1
	LIMBO	0.08%	3	2

Table V. Summary of clustering for the six SPEC benchmarks.

related functions improve the compilation time of inter-procedural optimizations and gathers similar run time performance. We have shown that optimizations do not need to apply to all functions. We have grouped the highly interacting program functions together and have shown optimizing those highly interacting program functions can improve compilation time.

The run time of partitioned versions of the six benchmarks has slight slow down in most of the benchmarks. The main reason for this slight slow down is optimizing only a subset of program functions. In the partitioned version, only the translation units which are the outcome of partitioning algorithms are optimized. The remaining functions are not optimized and compiled with the flag *O0*. Also the optimization techniques at local function and basic block level are not applied to those remaining functions.

VI. Conclusion

This paper introduces a new approach in program optimization. Instead of applying standard program optimization on entire program translation units, the approach focuses on a subset of translation units. The optimized translation units are created using profiling information.

Our approach is tested on six benchmarks. The compilation and run time of the partitioned version is compared to the original un-partitioned version. The compilation time improves in all benchmarks while having similar run time performance compared to the original benchmark version.

Our approach makes use of the standard optimization flag to optimize the translation units. One possible future research is the determination of different optimization settings for each translation unit using iterative optimization orchestration algorithms as depicted in the paper [8].

References

[1] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[2] *GCC, the GNU Compiler Collection*. [Online]. Available: http://gcc.gnu.org/

[3] K. Beyls and E. D'Hollander, "Refactoring for data locality," *Computer*, vol. 42, no. 2, pp. 62 –71, feb. 2009.

[4] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, p. 026113, Feb 2004.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, pp. 150–165, 2005.

[7] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, September 2006.

[8] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Proceedings of the International Symposium on Code Generation and Optimization. CGO 2006*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 319–332.