H-0247 (H0611-009) November 15, 2006
Computer Science

IBM Research Report

Aggressive Function Inlining with Global Code Reordering

Omer Boehm, Daniel Citron, Gadi Haber, Moshe Klausner, Roy Levin
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Aggressive Function Inlining with Global Code Reordering

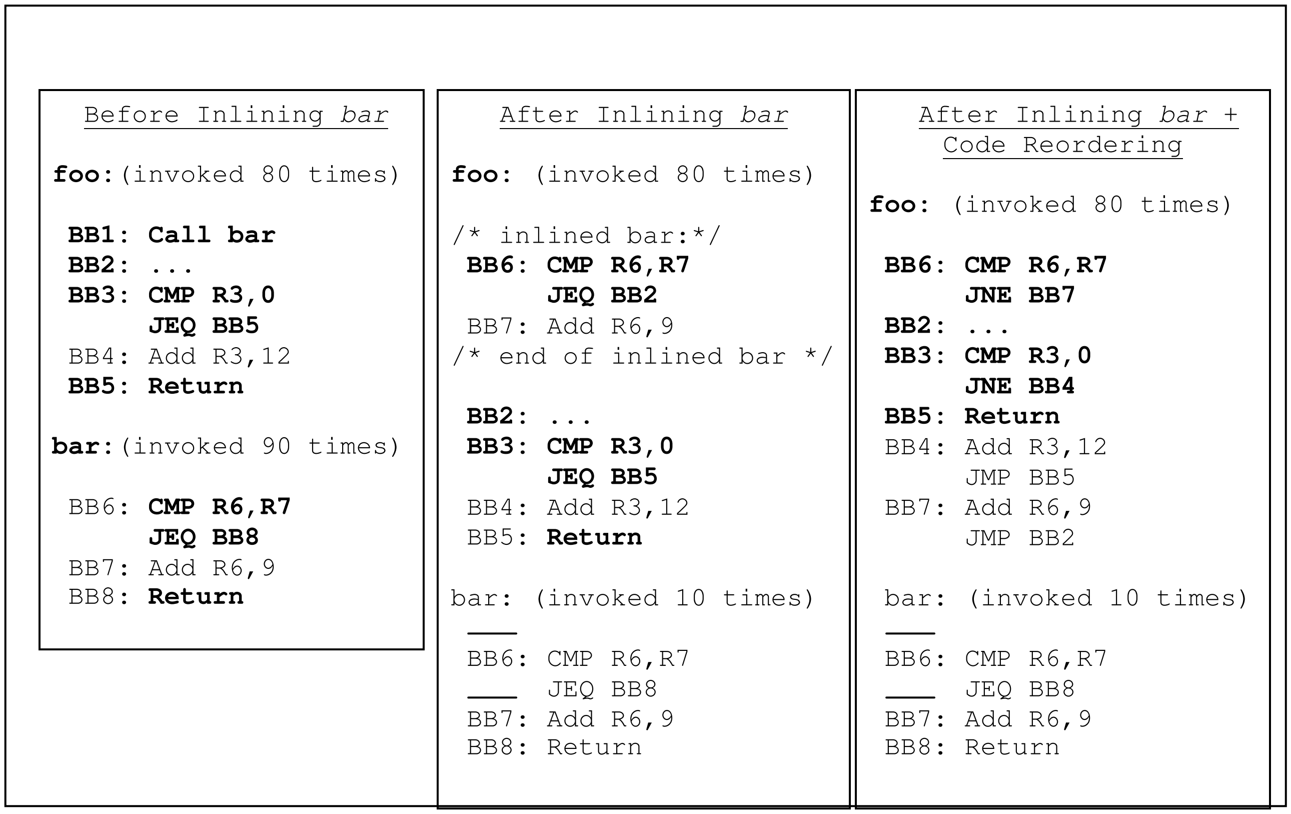
Omer Boehm, Daniel Citron, Gadi Haber, Moshe Klausner, Roy Levin
IBM Research Lab in Haifa, Israel
{omerb, citron, haber, klausner, royl}@il.ibm.com

Abstract

Code reordering and function inlining are commonly used today for improving memory locality and eliminating memory accesses. In this paper we present a new profile based optimization of an aggressive function inlining, based on a study of the mutual effects between code reordering and inlined functions. The proposed optimization duplicates frequently executed large functions and replaces their function calls by their bodies, while removing corresponding call and return instructions at the call sites. The inlining step is then followed by global code reordering to achieve better grouping of frequently executed code across function calls. The aggressiveness of the function inlining relies on the fact that no constrains are applied on the size, or the number, of the functions being inlined - only execution frequency is taken into consideration during the inlining process. The global code reordering that follows the function inlining guarantees the separation between cold and hot segments of the inlined functions.
The optimization was implemented into FDPR (Feedback Directed Program Restructuring), a post-link optimizer, which is part of the IBM AIX and Linux operating systems for the IBM pSeries servers. Our experiments were applied after the –O3 optimization phase of the compiler and results on SPECint2000 show an improvement of up to 8% (average of 4%) in execution time, on top of code reordering optimization, resulting from improving code locality and removing corresponding call and return instructions of inlined functions.

1. Introduction

Over the past several years, many methods and tools have been developed to improve application performance by improving the program's locality. Code reordering [3, 6, 7, 9, 10, 17, 19] and function cloning [8, 24-26] are known optimizations for improving code locality and I-cache behavior. Function inlining [1, 2, 4, 8, 11, 20-23] is another optimization that is heavily used by compilers today. In this paper we discuss the performance effects of inlining large frequently executed functions which are inlined at their calling sites, the way done today at the initial step of function inlining optimization. Thus, inlined functions in this work are not followed by applying any register allocation or code scheduling steps.. Therefore, avoiding the long compilation time associated with aggressive function inlining. Although commonly used today, function inlining can cause code bloat and degraded performance, forcing compilers to limit their scope to relatively small functions. Chen et al. [5] and McFarling [13] examined the effect of function inlining on cache behavior. They found that overall, function inlining seems to be largely ineffective on an optimized layout of basic blocks because the code expansion caused by inlining increases cache conflicts.



```
Before Inlining bar

foo:(invoked 80 times)

 BB1: Call bar
 BB2: ...
 BB3: CMP R3,0
      JEQ BB5
 BB4: Add R3,12
 BB5: Return

bar:(invoked 90 times)

 BB6: CMP R6,R7
      JEQ BB8
 BB7: Add R6,9
 BB8: Return
```

```
After Inlining bar

foo: (invoked 80 times)

/* inlined bar:*/
 BB6: CMP R6,R7
      JEQ BB2
 BB7: Add R6,9
/* end of inlined bar */

 BB2: ...
 BB3: CMP R3,0
      JEQ BB5
 BB4: Add R3,12
 BB5: Return

bar: (invoked 10 times)
 ___
 BB6: CMP R6,R7
 ___  JEQ BB8
 BB7: Add R6,9
 BB8: Return
```

```
After Inlining bar +
Code Reordering

foo: (invoked 80 times)

 BB6: CMP R6,R7
      JNE BB7
 BB2: ...
 BB3: CMP R3,0
      JNE BB4
 BB5: Return
 BB4: Add R3,12
      JMP BB5
 BB7: Add R6,9
      JMP BB2

 bar: (invoked 10 times)
 ___
 BB6: CMP R6,R7
 ___  JEQ BB8
 BB7: Add R6,9
 BB8: Return
```

(a) (b) (c)

Figure 1. Code Reordering Can Help Function Inlining

One of the issues related to inlined functions is the insertion of cold (rarely executed) code next to hot code. For example, consider Figure 1. Here we have a function *foo* which includes a hot call to another function *bar* shown in Figure 1a.. However, when we replace the call to *bar* by its inlined body in Figure 1b, we get a mixture of both hot and cold basic blocks (BB6, BB7, BB2). This causes an immediate negative effect on the I-cache behavior. One way to solve this problem is to apply global code reordering after function inlining, given in Figure 1c. As a result, all hot code is grouped together and the I-cache ratio improves.

Interestingly, the other way around is also true. Limitations in the code reordering optimization can be resolved by using function inlining. As an example, consider Figure 2. Here the hot path within function *foo* includes a hot call to function *bar* in Figure 2a. Therefore, ideally, the best code order would include the hot code preceding the function call to *bar* in *foo,* following by the hot code in *bar* itself and then back to the hot code proceeding the instruction call to *bar* in *foo,* as given in Figure 2b. Unfortunately, although this would be the ideal code order, there is an extra unconditional jump instruction that we are forced to add to the code right after the call instruction to *bar*. This jump instruction is necessary in order to maintain the original semantic of the program, so that the return instruction of *bar* will reach BB2. Applying function inlining before code reordering can solve this problem, as shown in Figure 2c. Here we can see that after inlining function *bar* at the call site in *foo,* the code reordering manages to pack the optimal hot path without the need for extra jump instructions.

In this paper we concentrate on the synergy between function inlining and global code reordering. We present an aggressive function inlining optimization based on edge profiling gathered on some representative workload. The inlining optimization is considered aggressive, as it is not limited to small sized functions and does not bound the resulting code size. Large functions are inlined, as well as small ones, according to their calling frequency. The suggested optimization performs function inlining of functions to each of its call sites that do not contain a hot directed path in the control flow graph between them. This is to avoid cache conflicts effects at run-time. We show that by inlining functions at call sites, which adhere to the above criteria, increases performance noticeably, provided the inlining is followed by a global code reordering. The code reordering groups all the hot code together and places all rarely executed code farther away in the code area. In this paper we describe the heuristics applied when choosing which functions to inline, along with the analysis of the effects on performance and the resulting code.

Although, the above technique can be applied at compiler level, we chose to implement it in a post-link

```
Before Code Reordering

foo:(invoked 80 times)

 BB1: Call bar
 BB2: ...
 BB3: CMP R3,0
      JEQ BB5
 BB4: Add R3,12
 BB5: Return

bar:(invoked 90 times)

 BB6: CMP R6,R7
      JEQ BB8
 BB7: Add R6,9
 BB8: Return
```

(a)

```
After Code Reordering

foo:(invoked 80 times)

 BB1: Call bar
      JMP BB2 (penalty)
Bar:
 BB6: CMP R6,R7
      JNE BB7
 BB8: Return
 BB2: ...
 BB3: CMP R4,R5
      JNE BB4
 BB5: Return

 BB7: Add R6,9
      JMP BB8
 BB4: Add R3,12
      JMP BB5
```

(b)

```
After Reordering +
    Inlining

foo:(invoked 80 times)

 BB6: CMP R6,R7
      JNE BB7
 BB2: ...
 BB3: CMP R4,R5
      JNE BB4
 BB5: Return

 BB7: Add R6,9
      JMP BB8
 BB4: Add R3,12
      JMP BB5

bar:(invoked 10 times)
 ___
 BB6: CMP R6,R7
 ___  JNE BB7
 BB8: Return
```

(c)

Figure 2. Function Inlining Can Help Code Reordering

optimization tool called FDPR (Feedback Directed Program Restructuring) reported in [9, 10, 16, 19]. Other known post-link tools can be found in [6, 7, 15, 18, 20]. FDPR was also used to collect the profile information for the optimizations presented here. The optimizations were applied on the SPECint2000 benchmark on the AIX Power4 64-bit platform. The performance results show an improvement of up to 11% and an average of 6.5% when applying function inlining followed by global code reordering and up to 8% with an average of 4% on top of code reordering.

This paper is organized as follows: Section 2 discusses related works. Section 3 describes the optimization method of the aggressive function inlining. Section 4 discusses the effects of the inlining on the code before and after applying code reordering. Experimental results are given in Section 5, followed by conclusions in Section 6.

2. Related work

As mentioned in the introduction, although function inlining can help improve performance, it can sometimes cause a substantial code bloat which results in a higher rate of I-cache misses and performance loss. As a result, inlining is often applied only to small functions. Nevertheless, several works have addressed the code expansion issue by suggesting additional heuristics which handle inlining of large functions.

Chen et al. [5] showed that code expanding optimizations have strong implications on instruction cache design. They found that function inline expansion improves the performance for small cache sizes, but degrades the performance of medium caches. McFarling [13] describes a method of determining which procedures to inline for machines with instruction caches. The method uses profile information as well as cache size and cache miss penalty information in order to define a relevant cache model. This cache model is then used to weigh the benefit of removing calls against the increase in the instruction cache miss rate for choosing inlining candidates. Our proposed solution does not rely on a cache model or any architectural information such as I-cache structure or code size constraints and, therefore, suitable for different machine configurations.

Ayers et al. [2] and Das [8], found an analogy between the code expansion problem and the Knapsack problem. They used this analogy to help in identifying appropriate candidates for function inlining. Arnold et al. [1] tried to find the best candidates that would fit to a given code size budget for the Jalapeño dynamic optimizing compiler. Triantafyllis et al. [21] suggested to use aggressive inlining together with code specialization techniques for the EPIC compiler. In their work they eliminated the need to limit the amount of code growth due to the specialization phase that reduces the amount of code being duplicated. The main difference of the above works from our proposed optimization is based on applying global code reordering after function inlining and by focusing on inlining functions only to call sites which are unlikely to increase cache conflicts as they are executed in different time frames. As a result, the heuristic of selecting candidates for function inlining does not depend on limiting the code size growth.

Way et al. in [22, 23] suggest different inlining heuristics that are based on the idea of “region based” analysis. The usage of regions is designed to enable the compiler to operate on reduced control flow graphs of inlined functions, before applying the register allocation and scheduling algorithms on them. The formation of a region is guided by profiling information and is similar to the way code reordering algorithm determines the new order of the basic blocks. In our work we concentrate on the specific optimization of code reordering that is applied after function inlining. As a result, we manage to show performance gain by concentrating on the study of the program locality effects of code reordering together with function inlining. Although the idea of compiling applications in a region based can help reduce compilation complexity, the heuristics for determining the function inlining candidates are also bounded by code size limitations, as opposed to this work.

For function inlining optimization, Way et al. [26] describe a new profile based method for determining which functions to inline in order to avoid cases of code bloat, by collecting path profiling information on the call graph of the program.

The post-link tools PLTO [20] and ALTO [15] address the issue of code inlining as part of their post-link optimizations. PLTO uses cache model for determining which functions to inline (similar to McFarling's [13]). We have also chosen to implement our techniques at post-link level using the FDPR tool [9, 10, 16, 19]. However, in our work we eliminate the restriction on the increase of code size by selecting only hot functions as candidates and by not duplicating them to call sites for which they are likely to cause cache conflicts.

3. Aggressive Inlining Method

The main issue of function inlining is that different copies of the same function can sometimes compete with one another in the I-cache. If a function is frequently called from different places in the code, duplicating it can cause higher cache miss ratio due to references to the multiple copies. For these cases, maintaining only the original copy of the callee function guarantees that function callers will share the same copy in the cache. In this work, for each potential inlining candidate function *foo* which is called by more than one frequent caller, we scan for hot paths between the callers, in order to determine if *foo* can be inlined without causing cache conflicts at run-time. In this work, in order to determine the frequency degree of the edges in the paths between the callers, an input threshold parameter, referred to as the *Hotness Threshold (HT)*, is used.

For example consider the following C code:

```
void jar1()
{
  for (i=0; i < MAX_NUM; i++) {
    foo();
    bar();
  }
}

void foo()
{
 gal();
}

void bar()
{
 gal();
}
```

In this example, inlining function *gal* into both *foo* and *bar* will most probably cause cache conflicts during the execution of *jar1* since both copies of *gal* will be frequently executed at the same time frame. However, in the following code example, inlining *gal* into *foo* and *bar* will not cause significant number of cache misses:

```
void jar2()
 {
   for (i=0; i < MAX_NUM; i++)
     foo();

   for (i=0; i < MAX_NUM; i++)
     bar();
}
```

The reason for this derives from the fact that there is a relatively cold edge in the control flow of *jar2* between the function call to *foo* and the basic block containing the function call to *bar,* and each function call is executed in a different time frame of the program execution trace. This means that inlining function *gal* will cause relatively small number of I-cache conflicts at run-time and therefore, makes a good candidate for inlining.

After inlining all the selected calling sites, a global code reordering is applied on the code. Therefore, properly updating the profiling information to reflect the changes in the control flow resulting from the massive function inlining, is vital for an effective basic-block reordering.

The proposed algorithm consists of three main steps:

1. Identifying all functions that are to be inlined according to the call graph, the control flow graphs of each function and the profile information of the original program.

2. Performing the actual inlining process for the selected functions from step 1.
3. Performing global code reordering on the program containing the inlined functions.

Given the call graph of the program, the proposed algorithm must also handle recursive function calls that are reflected by cyclic paths in the call graph. In order to handle cyclic paths, the algorithm must remove one of the edges in the cycle. Note that different inlining orders are created for different edges being removed from the graph, as can be seen from Figure 3. The figure shows an example of a cycle in the call graph of a given program, in which function *f* includes a call to function *i* that in turn calls to another function *j* which in turn calls back to *f*. The different possible inlining chains created by removing different edges in the call graph are shown at the lower part of the figure according to the following rules: if some function *foo* contains an inlined calling site to some function *bar*, then *bar* will be drawn beneath *foo* and slightly aligned to the right. If for some reason, *bar* is drawn directly beneath function *foo*, without being aligned to the right, then this would mean that both *foo* and *bar* were inlined into some other function *gal* containing the two calling sites to *foo* and *bar*. Figure 3 shows all possible inlining chains created by removing different edges in the *f* – *i* – *j* cycle. For example, removing the *j* - *f* edge, will cause function *j* to be inlined into *i* which in turn will be inlined into *f* (as shown in the figure). Therefore, it is important to search for the maximal directed acyclic graph representation of the given call graph. This problem is a variation of the feedback edge set problem [12]. The problem is NP-hard and the time complexity of the algorithm is exponential with respect to the number of edges in the largest strongly connected component of *G*. However, since in practice, the number of recursive functions which participate in the creation of cycles in a call graph is usually very small, the time complexity is sufficiently small.

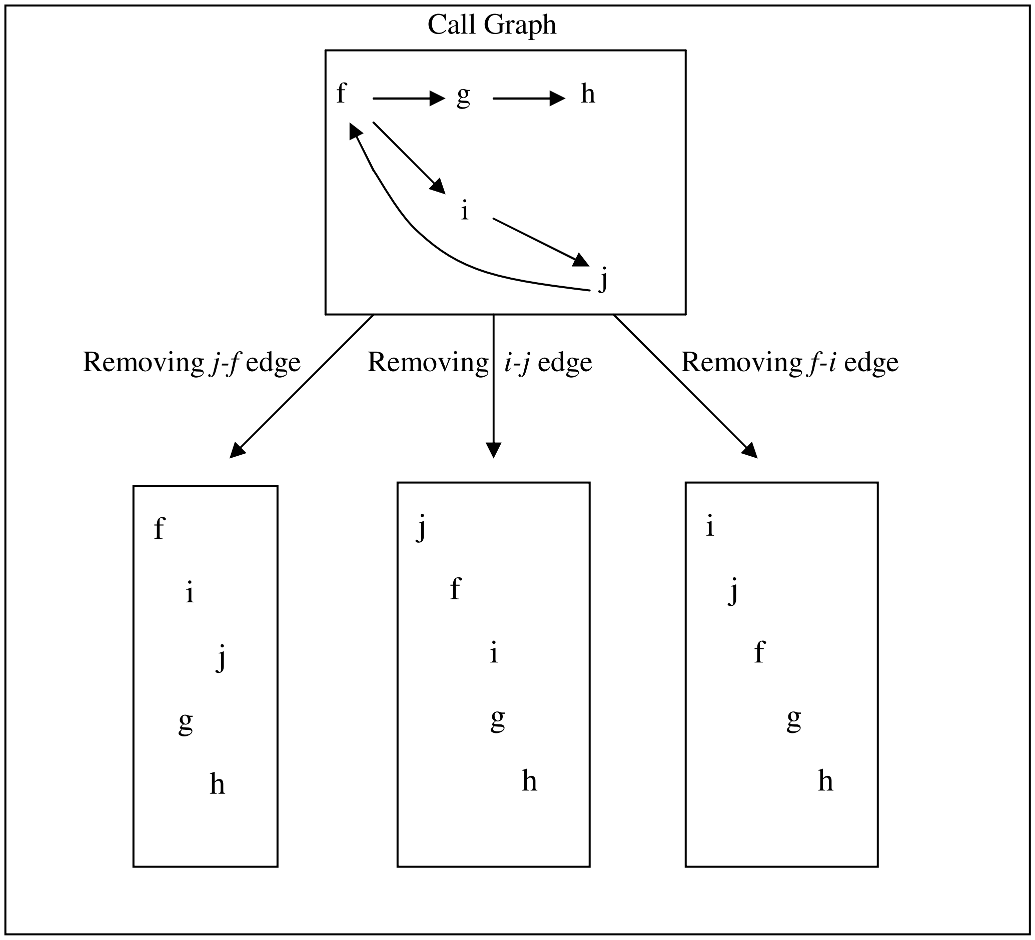


Figure 3 – Different Inlining Options for Cycles in the Call Graph

The main algorithm

1. Create the call graph *G* for the given program and attach a weight to each edge, representing the frequency of each function call according to the profile information.
2. Go over the call graph and remove any *cold* edges from *G*. In this work, a *cold* edge is defined as an edge in the control flow graph of the program whose weight falls below 10% of the average heat ratio *AvgHeat* which is calculated by the sum of all the frequencies of

all the executed instructions gathered during the profiling stage, devided by the total number of instructions in the program.

3. Remove cyclic paths from the resulted graph *G* by finding the smallest weighted set of edges *E* in *G* using the algorithm of Eades et. al. [12] for solving the feedback edge set problem.
4. Remove all the edges from *G* which can potentially cause cache conflicts when inlined at their call sites according to the filtering algorithm described further below.
5. Sort *G* in a topological order.
6. Go over all the functions in *G* according to their topological order starting from the functions in the roots in ascending order. For each function *f* :
    a. Duplicate the code of function *f* for each of the call sites which have an outgoing edge to *f*. All relevant information related to the copy of *f*, referred to here as *f_dup,* such as symbolic information, branch tables, traceback data etc., are to be duplicated as well.
    b. Update the call instruction to *f*, in all it call sites, to call *f_dup*.
    c. Update the weights of the edges in both call graph and control flow graph of both *f* and *f_dup* to reflect the changes resulting from duplicating *f*.
7. Go over all the functions in *G* starting from the leaf functions in descending order. For each function *f_dup* : Inline *f_dup* into its calling function *g* by placing its code within *g* and eliminating the save and restore instructions of the return address of *f_dup,* and by eliminating the call and return instructions in *f_dup*. Update the call and control flow graphs to reflect the changes resulting from inlining *f*.
8. Perform global code reordering on the resulted code in order to pack all hot code.

The above algorithm relies on the code reordering phase to improve the code locality after function inlining. Therefore, it is very important to update the profile information in both the call graph and control flow graph used by code reordering. If function *foo* is to be inlined into function *bar* then the profile information of both the inlined version of *foo* and the original *foo* function must be updated to describe the new state. The more the fix is representative of the state, the better the reordering results would be. In the absence of path profiling we calculate the ratio between the callee's prolog and the caller site count and fix all basic blocks' and edges' counts of *foo* by multiplying them with that ratio.

The Filtering Algorithm

Let *AvgHeat* be the average edge weight in the control flow graph of the entire program as calculated in step 2 of the main algorithm, and let *MaxHeat* denote its maximal weight. Let *HT* denote an input hotness threshold percentage between 0 and 1. The *Normalized Hotness Threshold* value *NormHT* is calculated according to the following formula:

For $HT = 0\%$, $\mathbf{NormHT = 0}$

For $HT = 100\%$, $\mathbf{NormHT = MaxHeat + 1}$

For $0\% < HT < 100\%$, $\mathbf{NormHT = \min(AvgHeat/\ln(1/HT^2), MaxHeat+1)}$

1. Every edge of the control flow graph, which falls bellow *NormHT* as calculated in step 1 above, is removed from the graph. As a result, the higher the input *HT* percentage, the more aggressive the optimization is. For *HT* = 0% all the edges in the call graph will be removed completey, thus disabling the optimization. For *HT* = 100% the call graph is left unchanged, enabling the optimization for every non-cold edge which was left in the call graph after applying step 2 of the main algorithm.
2. For each inlining candidate function *f* in the call graph *G*:

a. For every two incoming edges *e1, e2* to *f* let *caller1* be the caller basic block terminating with *e1* and let *caller2* be the caller basic block terminating with *e2*, and let fallthru1 and falthru2 be the basic blocks following *caller1* and *caller2* respectively.

b. Go over the control flow graph and search for directed paths from *fallthru1* to *caller2* and from *fallthru2* to *caller1*.

c. If both paths exist, then remove the *e1* and *e2* edges from *G*.

4. The Effects of Aggressive Inlining on the Program Code

The aggressive inlining process produces a set of “chains” of inlined code in the program. Each chain is a sequence of inlined functions contained within a single chain header function. A header function *f* may contain several inlined call sites to different functions, which in turn may include other inlined functions in them. All these inlined functions form a single chain which begins and ends with *f’s* prolog and epilog in respect. We claim that a chain correlates to one hot execution path through the program code in the combined call graph and control flow graphs of the inlined functions in the chain. Therefore, the optimization is aimed at creating large chains that will include as few hot function calls as possible in them. There are two types of chains produced after aggressive function inlining: “regular chains” which begin/end with a prolog/epilog of a cold header function, and “service chains” which begin/end with a hot prolog/epilog and are being called from other two or more chains. Figure 4 illustrates the code produced after aggressive inlining. The code is partitioned into a set of regular and service chains, service functions and isolated cold functions.

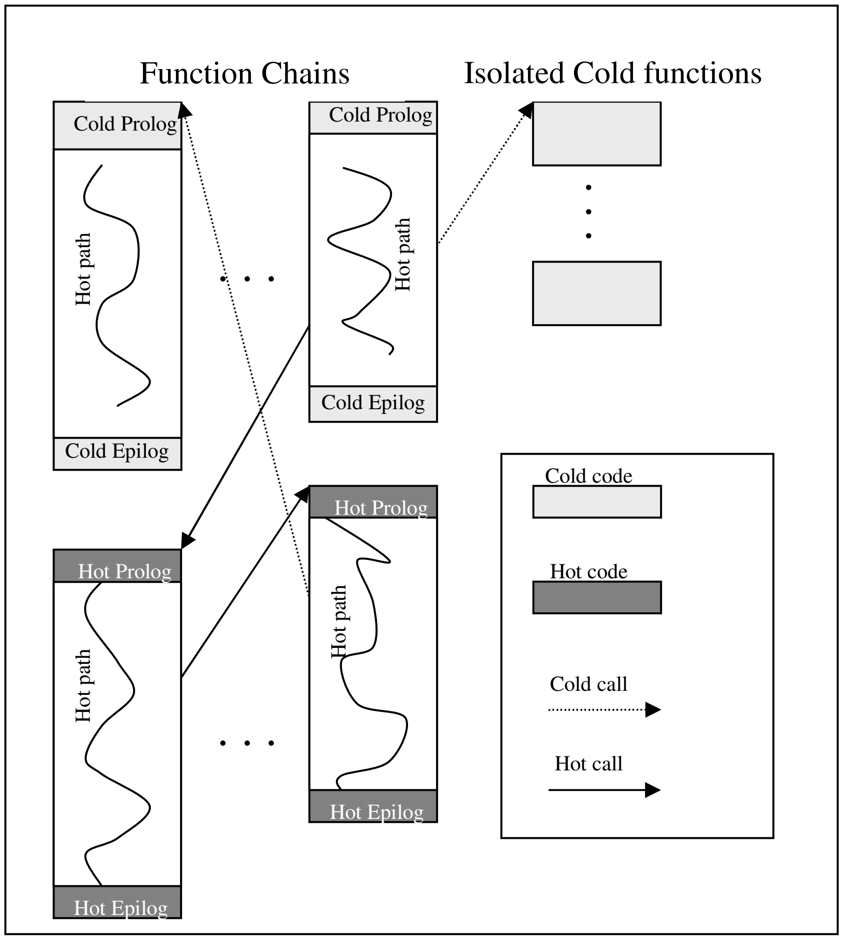


Figure 4. Code Layout After Function Inlining

The aggressive inlining has also an impact on the code reordering optimization potential. The heuristic of the code reordering algorithm for generating optimized sequences of basic blocks is based on the tracing idea. The algorithm starts with an entry point basic block and tries to grow a "trace" of basic blocks, using the profiling information. Another decision which the code reordering algorithm makes is when to stop growing a trace. Basically, when the probability of transferring the control to a next block in a trace falls below a certain frequency threshold, the algorithm decides to finish growing a trace and starts a new one. Note that in general, traces do not cross procedure boundaries due to the overhead caused by the need to add extra jump instructions during the code positioning phase, as was shown in Figure 2b. Another decision is where to start the next trace, as there are several alternatives for starting a new trace. In our code reordering algorithm we do not necessarily continue to cover the rest of the traces of the same procedure A. Rather, the algorithm can switch to a trace from another procedure C, then from D, and so on.

Note that the longer the traces produced by code reordering, the better program locality is. We claim that the average size of traces created before aggressive inlining vs. the average size after inlining can also serve as a measure for the improvement in program locality. In general, the average size of traces increases due to function inlining. This is due to the fact that traces that started to grow in a certain procedure can now grow into the corresponding inlined callee procedures, as the corresponding function call instructions are removed during function inlining.

5. Experimental Results

In this section we demonstrate the benefits of the aggressive inlining presented in the paper and try to explain the reason for the performance gain. The experiments were conducted on an IBM 64-bit Power4 Regatta machine with four processors, containing an I-cache of 16KB instruction, and a unified 4MB L2 cache. As mentioned in the introduction, the optimization was implemented into the IBM FDPR post-link tool, running on the AIX operating system version 5.2. The optimization method was applied on the SPECint2000 benchmark executable files, after being compiled as 64-bit programs with the IBM xlC compiler using the -O3 optimization flag. The profile information was gathered on the train input, whereas performance measurements were taken with the ref input. The function inlining optimization was applied with four different hotness threshold factors of 30%, 50%, 70% and 100% and included only basic code transformations of eliminating the corresponding call and return instructions.

Figure 5 demonstrates the synergy effect of function inlining together with code reordering. As can be seen from the figure, the performance improvement resulted from applying both optimizations is always higher than the sum of the performance gains when applied separately. In most cases, aggressive inlining, without code reordering, reduces the performance, as is the case with *gzip, gcc*, *crafty*, *parser*, *eon, gap vortex* and *bzip2*.

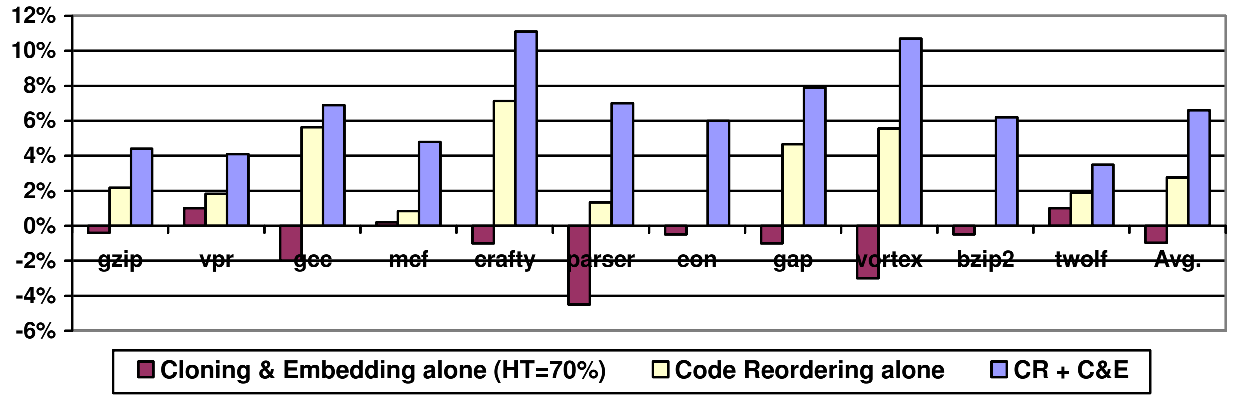


Figure 5 – The Synegry of Function Inlining with Code Reordering

Figure 6 shows the performance improvement on SPECint2000 measured after aggressive inlining followed by global code reordering, versus the performance after code reordering alone without function inlining. We can see that the performance improvement on top of code reordering reaches up to 8% with an average of 4% for *HT* = 70% which, in general, gives the best results.

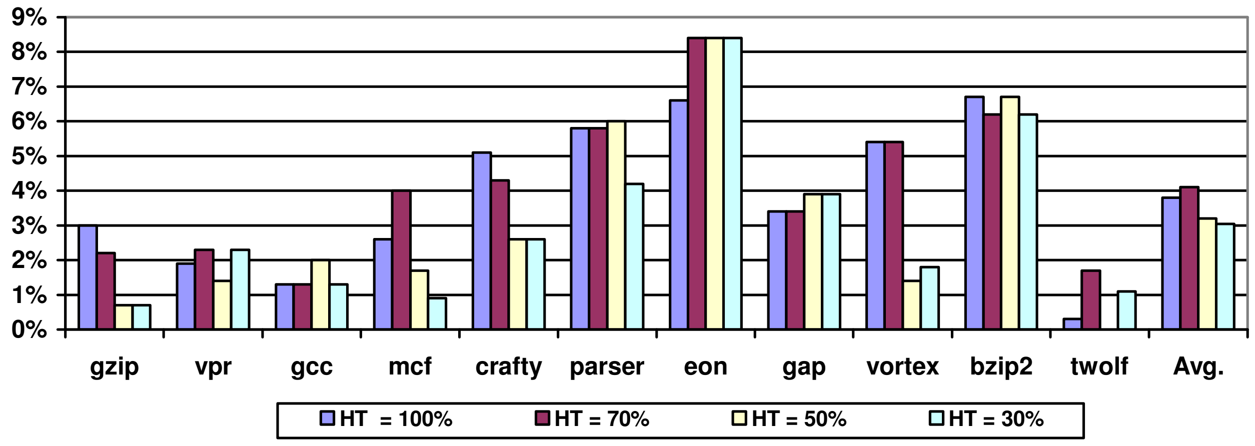


Figure 6 - Run-Time Improvement on top of Code Reordering (%)

Figure 7 shows the change in hot code size after function inlining. Interestingly, the size of the hot code in *gzip* and in *vortex* actually decreases for *HT* values of 30% or 50%, after applying function inlining. This is mainly due to the following two reasons:

1. The removal of call and return instructions together with their corresponding store and restore instructions of the returning address for each inlined function.
2. In most cases the original copy of each inlined function turns cold while only its inlined copy remains hot. Therefore, in such cases, function inlining did increase the total hot code size.

Note that since functions are inlined only for callers which are unlikely to create cache conflicts, as they are executed in different time frames, increasing the hot code footprint will not increase the number of cache conflicts in a noticeable manner.

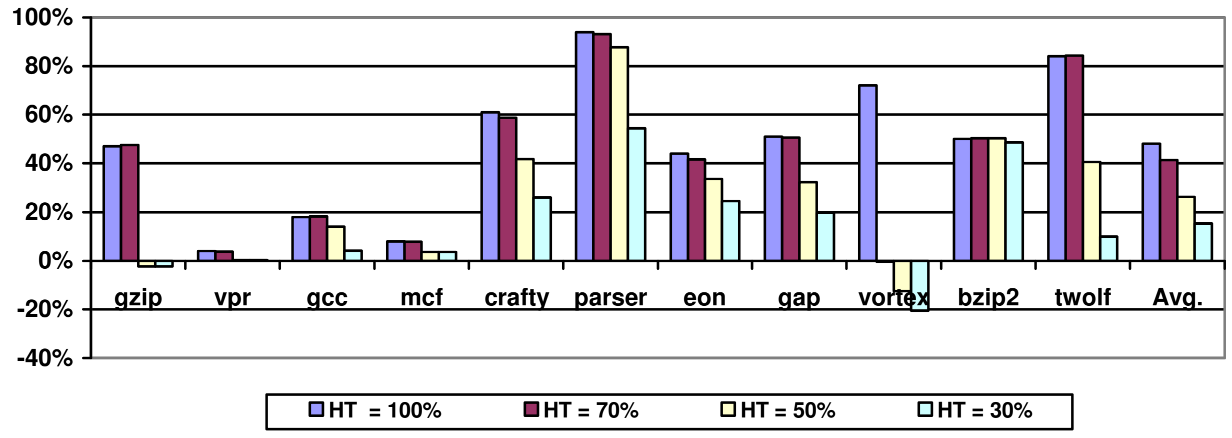


Figure 7 – Hot Code Size Expansion (%)

Figure 8 displays the total size expansion in the executable file. As can be seen from the figure, the executable size expansion is between 20% – 30% depending on the hotness threshold value being used, and can reach up to 120% with *vortex* when enabling all the non-cold call sites to be optimized (*HT* = 100%). It is interesting to see that the *eon* benchmark, which showed the highest performance improvement of 8% on top of code reordering in Figure 6, shows an exceptionally small increase in the total executable size after applying the optimization.

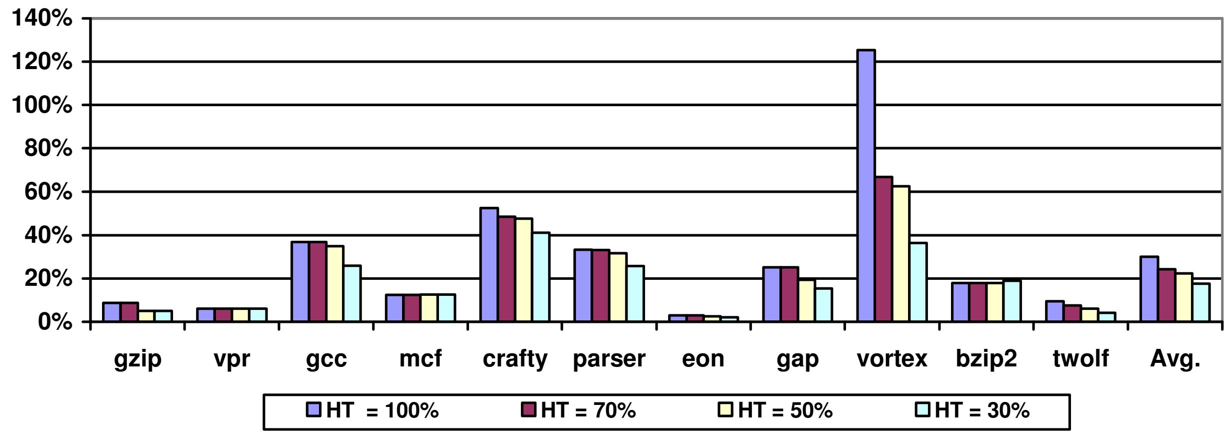


Figure 8 – Total Executable Expansion (%)

Figure 9 shows the total number of chains created after function inlining for *HT* = 70% and the ratio between chains starting/terminating by cold prologs/epilogs versus chains wrapped by hot prologs/epilogs (service chains). There is a correlation between the total number of chains and the high ratio of chains that are wrapped by cold prologs. A cold-prologed/epiloged chain indicates a complete trace in the program which contains all hot paths in it. Therefore, we would always thrive to reach a high percentage of cold-prologed chains in the program. In the figure, *crafty* and *eon* show the best ratio.

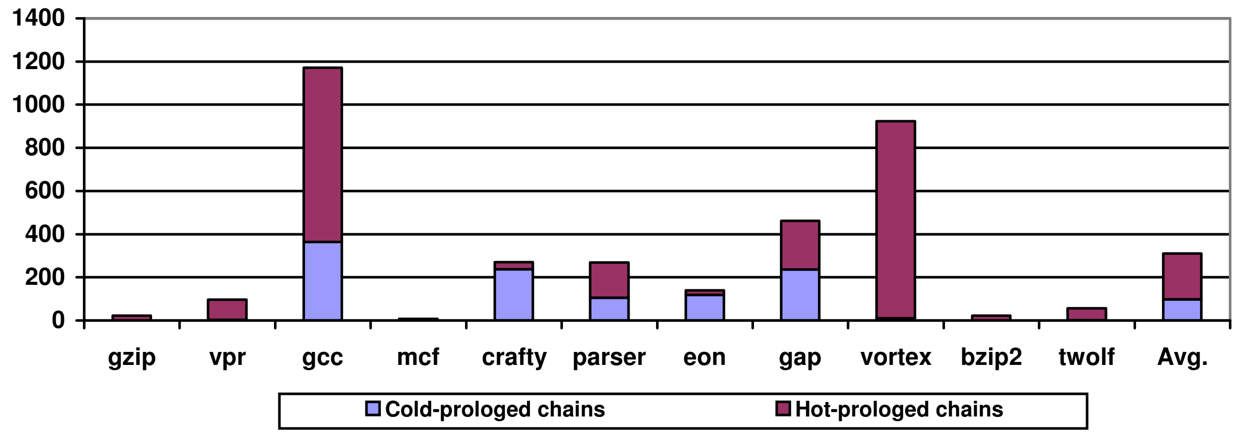


Figure 9 – Total Number of Inlined Chains for *HT* = 70% (in bytes)

Figure 10 shows the ratio between the average size of cold-prologed chains versus service chains. On average service functions which are called from several hot sites form a very small portion of all the hot chains in the optimized program. Overall, the average size of service functions which are called from several hot sites, form a very small portion of all the total average size of the hot chains in the optimized program.

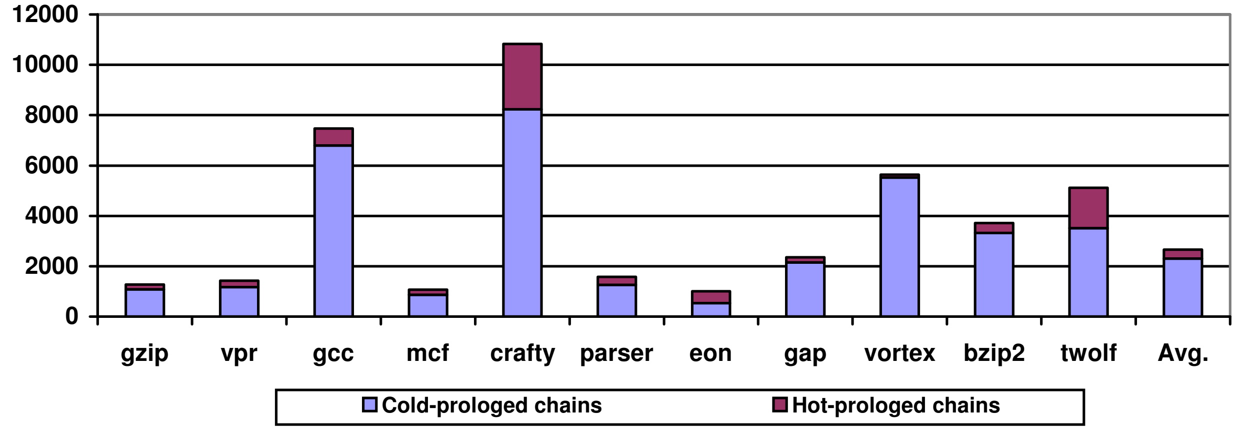


Figure 10 – Average Size of Inlined Chains for *HT* = 70% (in bytes)

Figure 11 shows the increase in the average size of the traces produced by code reordering preceded by function inlining, versus the size of the reordering traces without applying inlining. A trace size is computed by the sum of all its instructions. An average trace size is the average size of all the hot traces for which their average execution count is above the given hotness threshold. On average, code reordering traces have increased between 20% - 50% depending on the hotness threshold value being used. As explained in Section 4, the main reason for the growth derives from the ability of the reordering algorithm to build traces which cross procedure boundaries after function inlining.

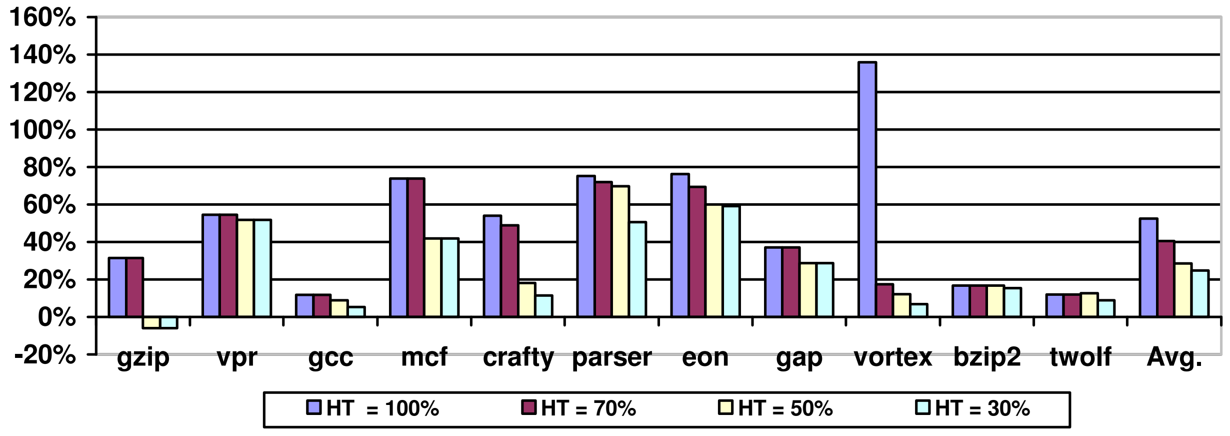


Figure 11 – Expansion of Average Trace Size (%)

The experimental results show that inlining together with code reordering helps increase the locality by eliminating cold code as well as increasing the traces in the hot code. In some cases the code locality is the main contributor, in others, the increase in trace size, and in others the combination of the two. From the experiments we also learn that there are several factors for choosing the most suitable *HT* percentage factor: the size of the cache, the percentage of growth of the average trace size during code reordering, the ratio of hot service chains versus cold regular ones, and the increase/reduction in hot code size. In our experiments all the SPECInt2000 benchmarks along with their optimized versions, did not exceed the L2 cache capacity of 4MB. For example, *mcf* benchmark which has a small size code, shows the best performance results even with a threshold factor of 100% which is equivalent to inlining any non-cold call site. The *mcf* is an example where the large cache size has a dominant affect on its performance. On the other hand, the *eon* benchmark shows a decrease in performance when using *HT* = 100% versus *HT* = 70%. The *eon* benchmark, which is 4 times larger than the size of *mcf*, already becomes too large to fit into the cache for *HT* = 100% and a more appropriate filtering must be used. In general, an *HT* value between 70% - 80% showed a peak performance also in commercial appliactions.

6. Conclusions

In this work we presented a new optimization based on a study of the mutual effects between function inlining and code reordering. The proposed optimization performs an aggressive inlining which is not bounded by the increase in code size. The inlining is proceeded by a global code reordering phase. As a result, the increase in total code size and in the hot code footprint does not degrade performance due to the fact that execution time frames are taken into consideration.

The performance results measured here did not include any additional optimizations which are part of the function inlining optimization, such as register allocation, instruction scheduling etc.

7. References

[1] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney "A comparative study of static and profile-based heuristics for inlining" *In Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pp 52-64, 2000

[2] A. Ayers, R. Gottlieb, and R. Schooler, "Aggressive Inlining", In *Proceedings of the '97 ACM SIGPLAN conference on Programming language design and implementation*, Pages 134-145, Las Vegas, June 1997

[3] B. Calder, and D. Grunwald, "Reducing Branch Costs via Branch Alignment", *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 242-251.

[4] P. P. Chang, S. A. Mahlke, W.Y. Chen, and W.W. Hwu, "Profile-guided automatic inline expansion for C programs", *Software Practice and Experience,* Vol, 22, pp. 349-370, May 1992

[5] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu. "The Effect of Code Expanding Optimizations on Instruction Cache Design", *IEEE Transactions on Computers*, 42(9):1045-1057, September 1993.

[6] R. Cohn, D. Goodwin, and P. G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike", *Digital Technical Journal*, vol. 9, no. 4, Digital Equipment Corporation 1997, pp. 3-20.

[7] R. Cohn, and P. G. Lowney, "Hot Cold Optimization of Large Windows/NT Applications", *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, December 1996, pp. 80 - 89.

[8] D. Das, "Function Inlining versus Function Cloning", *ACM SIGPLAN Notices,* ACM Press, New-York USA, Volume 38, Issue 6, June 2003, pages 23 - 29

[9] G. Haber, E.A. Henis, and V. Eisenberg, "Reliable Post-link Optimizations Based on Partial Information", *Proceedings of the 3rd Workshop on Feedback Directed and Dynamic Optimizations*, December 2000.

[10]E. A. Henis, G. Haber, M. Klausner and A. Warshavsky, "Feedback Based Post-link Optimization for Large Subsystems", *Second Workshop on Feedback Directed Optimization*, Haifa, Israel, November 1999, pp. 13-20.

[11]W.W. Hwu and P. P. Chang, "inline function expansion fro compiling C programs", in *proceeding of the '89 ACM SIGPLAN conference on Programming Language Design and Implementation,* pp. 246-257, June 1989

[12]P. Eades, X. Lin, and W. F. Smyth. "A fast and effective heuristic for the feedback arc set problem". *Info. Proc. Letters*, 47:319-323, 1993.

[13]S. McFarling. "Procedure Merging with Instruction Caches", *In proceedings of the SIGPLAN 1991Conference on Programming Language Design and Implementation,* pages 71-79, June 1991.

[14]S. S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, San Francisco, California, 1997.

[15]R. Muth, S. Debray, S. Watterson, "alto: A Link-Time Optimizer for the Compaq Alpha", Technical Report 98-14, Dept. of Computer Science, The University of Arizona Dec. 1998.

[16]I. Nahshon and D. Bernstein. "FDPR - A Post-Pass Object Code Optimization Tool", *Proc. Poster Session of the International Conference on Compiler Construction*, pp. 97-104, April 1996.

[17]K. Pettis and R. Henson, "Profile Guided Code Positioning", *Proc. Conf. on Programming Language Design and Implementation*, June 1990, pp. 16-27.

[18]T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad. and B. Chen, "Instrumentation and Optimization of Win32/Intel Executables Using Etch", *Proceedings of the USENIX Windows NT Workshop.* August 1997, pp. 1-7.

[19]W. J. Schmidt, R. R. Roediger, C. S. Mestad, B. Mendelson, I. Shavitt-Lottem, and V. Bortnikov-Sitnitsky, "Profile-directed Restructuring of Operating System code", *IBM Systems Journal*, 37, No. 2, 1998, pp. 270-297.

[20]B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A link-Time Optimizer for the Intel IA-32 Architecture", In *Proceedings of Workshop on Binary Rewriting,* Sept 2001

[21]S. Triantafyllis, M. Vachharajani, and D. I. August "Procedure Boundary Elimination for EPIC Compilers" In *Proceedings of the Second Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology*, November 2002.

[22]T. Way, B. Breech, and L. L. Pollock, "Region Formation Analysis with Demand-driven Inlining for Region-based Optimization", PACT, pages 24-36, 2000

[23]T. Way and L. Pollock, ``Evaluation of a Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler," *IASTED International Conference on Parallel and Distributed Computing and Systems* (PDCS 2002), Cambridge, Mass., November 2002

[24]M. W. Hall., "Managing Interprocedural Optimization", Ph.d. thesis, Rice University, April 1991

[25]K. D. Cooper, M. W. Hall and K. Kennedy, "A methodology for procedure cloning ", *Computer Languages*, pages 105-117, 1993.

[26]T. Way, B. Breech, W. Du, V, Stoyanov, and L. Pollock, "Using Path-pectra-based Cloning in Regional-based Optimization for Instruction Level Parallelism", ISCA 14th International Conference on Parallel and Distributed Computing Systems, (ISCA PDCS), pp. 83-90 2001