A FUNCTIONAL LEVEL MEMORY OPTIMIZATION TOOL – THE SLICER

By

Lizhuang Zhao

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

(COMPUTER SICENCE)

RENSSELAER POLYTECHNIC INSTITUTE

TROY, NEW YORK, USA, 2004

Abstract

In this thesis, we present a memory optimization tool – the Slicer, which is used for optimizing large long-time running programs. According to calling relationship among functions of a program, the Slicer relocates these functions in the memory, so as to decrease the code page faults invoked by the functions' callings. The relocation follows three basic rules. First, functions that are often called together should be put into the same page, so at most one page fault is enough to call a group of this kind of functions. Second, functions with great calling frequency should be placed together, such that a page fault for one of them will load other functions frequently being called into the memory. Third, we should avoid letting a function to span two pages unless its size is more than a page size, so loading it only incurs one page fault instead of two or more.

The Slicer works at the source code level, and consists of three basic components: the Instrumenter, the Splitter and the Stuffer, which are created mainly using Lex, Yacc and Perl. The Instrumenter collects the functions' past calling sequence for predicting the future behavior of the program. The Splitter is used to split functions from the program files, in order to move them freely in linking stage. The Stuffer inserts some small blank functions to prevent a function from spanning two pages, so as to decrease the page faults further. We also use data mining techniques to analyze the past calling patterns of functions and generate the new memory layout. Finally, we choose a typical network application Apache HTTP server as our experimental object to exemplify a long-time running program with open source code that shows similar behaviors over a considerable period of time.

Table of Contents

1. Introduction

1.1 Motivation

For each program, there are specific calling relationships among its functions each of which has its own utilization frequency. This information is useful for execution optimization of programs, especially the long-running ones that show similar behavior over a considerable period of time. In this project, we realize this performance optimization opportunity by using a memory layout at program functions' level that should achieve better performance (less code page faults) than random order of functions' linking. To this end we created an optimization tool, called the *Slicer*. Our goal is to decrease the code page faults and therefore to increase the average response time of the whole system.

1.2 Background

Code optimization techniques can be categorized into two classes: static and dynamic. Static code optimization is done during the compile-time period when converting the source code into machine code. There are some general techniques for static optimization, such as loop unrolling, hot-cold optimization, etc. Loop unrolling attempts to unroll certain innermost loops, minimizing the number of branches and grouping more instructions together to allow efficient overlapped instruction execution [Huang97]. Hot-cold optimization is a technique that uses profile information to partition each routine into frequently executed (hot) and infrequently executed (cold) parts. By removing unnecessary operations in the hot portion and adding compensation code on transitions from hot to cold if necessary, better performance of program execution is achieved. For hot-cold optimization, it is possible that transformations optimize a particular region of code at the expense of other regions [Cohn96]. These two, and many other static methods focus on the assembly level optimization, and achieve a good performance by executing fewer instructions.

Dynamic code optimization performs the optimizations at run-time, which makes it more adaptive and flexible than static optimization. There are three phases for dynamic optimization system. First is to collect the behavior information of the application. Next, according to the information collected, prediction of the future behavior pattern of the

program is formed. Finally, the optimization operations are dynamically implemented to take advantage of the predicted patterns. These steps may repeat from time to time to let the executable file accommodate to the current situation [Cohn97], [Franz97]. The compiler enables the continuation of the optimization process by generating an executable capable of monitoring itself and performing its own optimizations at run time. However, this imposes an additional execution time on the monitored process. Moreover, often it is difficult for the dynamic optimization to predict the future patterns.

1.3 Thesis Organization

This thesis is composed of 7 chapters. It starts with a brief introduction on the Slicer's motivation and related background. Second chapter gives the detailed working principles of the Slicer. Those are based on the aspects that we could improve by changing the functions' layout. The third chapter describes the overall architecture of the Slicer and its three components: the Instrumenter, the Splitter and the Stuffer. Next, by applying the Slicer to a concrete example, the Apache HTTP server, we demonstrate the Slicer's working process and identify some issues that need to be considered in future works. After that, we give and analyze the experimental results from the Apache server testing. In the end, we conclude and address what possibly need to be improved in the future.

2. Working Principles

2.1 Observations

The functions in a program do not work separately and there is some calling relationship among them. In this project, we study this kind of relationships to lower page faults and so achieve better performance. The functions calling relationship in a program led us to the following observations. First, the calling frequency of each function is different, generally. By collecting those functions that are called frequently together into one page, we decrease the page faults and cache misses. Second, some functions may always be called one after another in a consecutive way. By grouping them into the same page, we decrease the page faults and cache misses. Third, avoiding spreading one function over two pages decreases the page faults and cache misses. In order to achieve this we need to use some small blank functions to fill the end fragment of each page. Such fillings increase the final executable file size a little. However, our experience indicates that the size enlargement is less than 5%. To utilize the above observations, data mining technique can be used to discover the potential performance optimizations. For example, frequent patterns mining is suitable for case one, and sequence mining is good at case two, etc.

2.2 Object Selection

Based on the above observations, when selecting the optimization object, we need to consider the following rules. At first, the size of the software package should be large enough to achieve considerable performance, since small size will result in zero or less page faults (the whole executable always resides in the memory). Second, the software should be a long-running program; only then the proposed optimization can give significant benefits, i.e. decrease the frequently called functions' page faults. For a transitory application, most functions are called only a few times, so the page faults saving is negligible compared with the optimization process. Furthermore, the application should display similar behaviors over a considerable period of time, so it is possible for us to make a reasonable prediction of the application behaviors. If the program acts irregularly most of the time, then correct prediction is impossible in theory. Based on these rules, we chose Apache HTTP server on Linux platform, a famous open-

source HTTP server, as our optimization object. It is a long-running (e.g., several years) program with considerable size. Moreover, Apache HTTP server shows very similar action patterns over a period of time, because such a server always serves a relatively fixed group of people who share pretty similar interests and have routine access patterns. For example, most users of www.rpi.edu may be its students, and they like to access the local news and weather most often. All these similar access patterns assure our prediction to be valid and effective.

2.3 Pattern Collection

We need a C source level instrumentation tool to get the calling pattern. This tool will insert two different commands (unique for each function) at the start and end point of each function of the software package. As the executable runs, the running sequence of functions will be recorded into a file, which will be analyzed by a data mining module. For example, this is a piece of calling sequence:

fun1_start fun2_start fun2_end fun3_start fun3_end fun1_end fun4_start fun4_end …

From this sequence, we can see that it is possible that fun1, fun2 and fun3 always happened together. This assumption needs to be confirmed by the statistical and sequence analysis of large volumes of calling sequence data.

2.4 Pattern Analysis

The goal of pattern analysis is to decrease the page fault of a long-running executable program with considerable size, which is composed of tons of functions. The

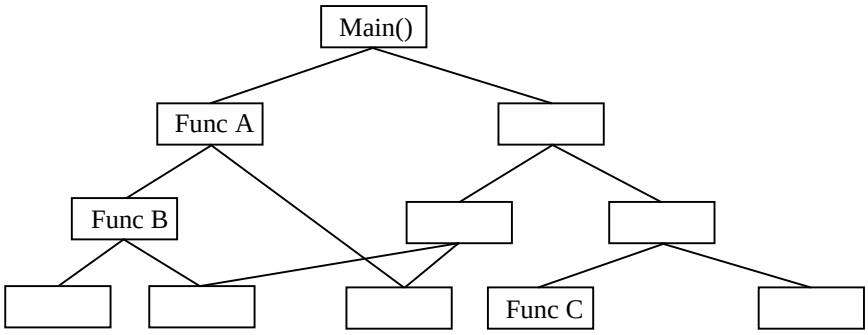

Figure 1. Example of a function-calling graph

basic organization of these functions is a layer structure (as in Figure 1) that is not necessarily a tree. From Figure 1, we can see that function A and function B may always be called together. However function B and function C have no calling relationship here. If we put those functions that are usually called one after another (in a very short time) together in a memory page, then the code page faults will drop a lot especially for those large long-running executables. We use data mining techniques to analyze the relationships among different functions of a code. In order to realize this, we developed an instrumentation tool, the Instrumenter, that modifies the source code to record the function calling sequence. The sequence then will be analyzed by data mining module to figure out the function groups in which all functions are often called together.

2.5 Functions Relocation

Next we need to group the related functions into the same page. For example, in Figure 2, three different functions (e.g., always called consecutively) located in three different pages are grouped into the same page, and if the left space is not large enough to hold another program function, it is stuffed with a small blank function to achieve page alignment. Page alignment will avoid a function to span two pages. So when a page fault happens for the function, it involves only one page fault, not two compared to the no stuffed case.

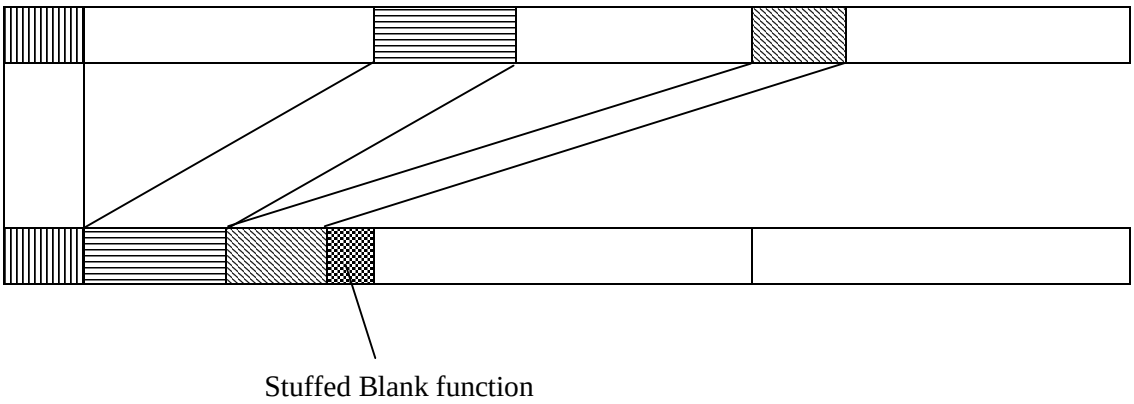

Figure 2. A group of three related functions

Sometimes it is possible that a function's size will be greater than a page size (4096 for our testing platform), i.e., one page cannot fully hold this function. In such a case, we will arrange the function in such a way that it spans as few different pages as possible.

For instance, in Figure 3, the original memory layout of an executable file is function1, function2, function3 and function4, as *row-1* shows. We relocate them to the order of function2, function4, function3 and function1, and let them to be page aligned. Then, the relocated memory layout becomes *row-2*, where 'b' represents the stuffed blank function for page alignment. In addition, *row-3* shows the boundary of each page in memory. From Figure 3, we can see that function 2 originally spans three pages but it spans only two pages after optimization. The function 3 originally spans only one page after optimization, compared to two before. So, when the system needs to fetch function 2 from the hard disk, only two page faults are needed, compared to three page faults before optimization. For function 3 and function 4, one page fault is needed for each.

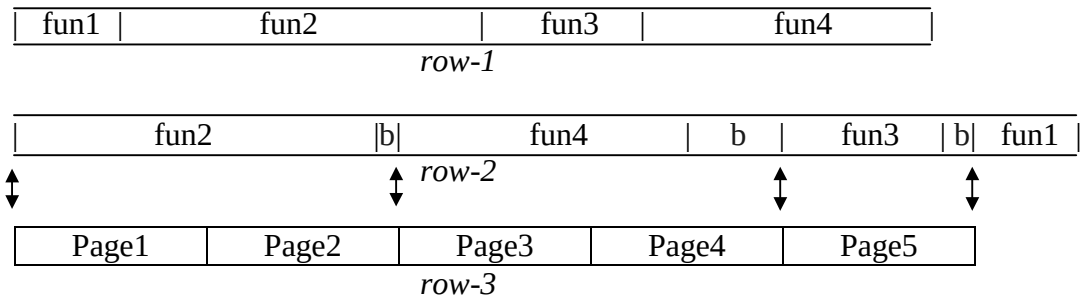

Figure 3. Page stuffing reducing cases in which a function spans pages

2.6 Coordinated Work

In order to realize the proposed optimizations, we implemented a tool, called the Slicer, that consists of three components: the Instrumenter, the Splitter and the Stuffer. The Instrumenter and the Splitter are created with Unix tools Lex and Yacc [John92]. The Instrumenter records the calling patterns of the program to be optimized and passes them to the data mining module. The Splitter is in charge of splitting each file of the program's source code into such different files that each of them contains only one function. As a results, we cac rearrange each function's relative position in the memory during linking stage. The Stuffer is primarily written in Perl language. It can calculate each blank function's position and corresponding length according to the optimized result from data mining module. It can also insert the stuffing functions to the page-ends when linking the final executable file. Figure 4 shows an example of the coordinated work between processes of the Splitter and the Stuffer. Initially, there are three C source files:

file1, file2 and file3. File1 contains functions fun1 fun2 and fun3. File2 contains functions fun4 and fun5. File3 contains functions fun6, fun7, fun8 and fun9. First, the Splitter splits the three files into 9 files, such that each file contains exactly one function. After that, we change their linking order according to the new memory layout sequence (e.g., 6-8-4-2-5-7-9-3) provided by the data miner. Finally, the Stuffer generates some blank functions and stuffs them into the rebuilt executable file so each function be page aligned at its start point in the memory. The obtained executable file is our final optimized binary application.

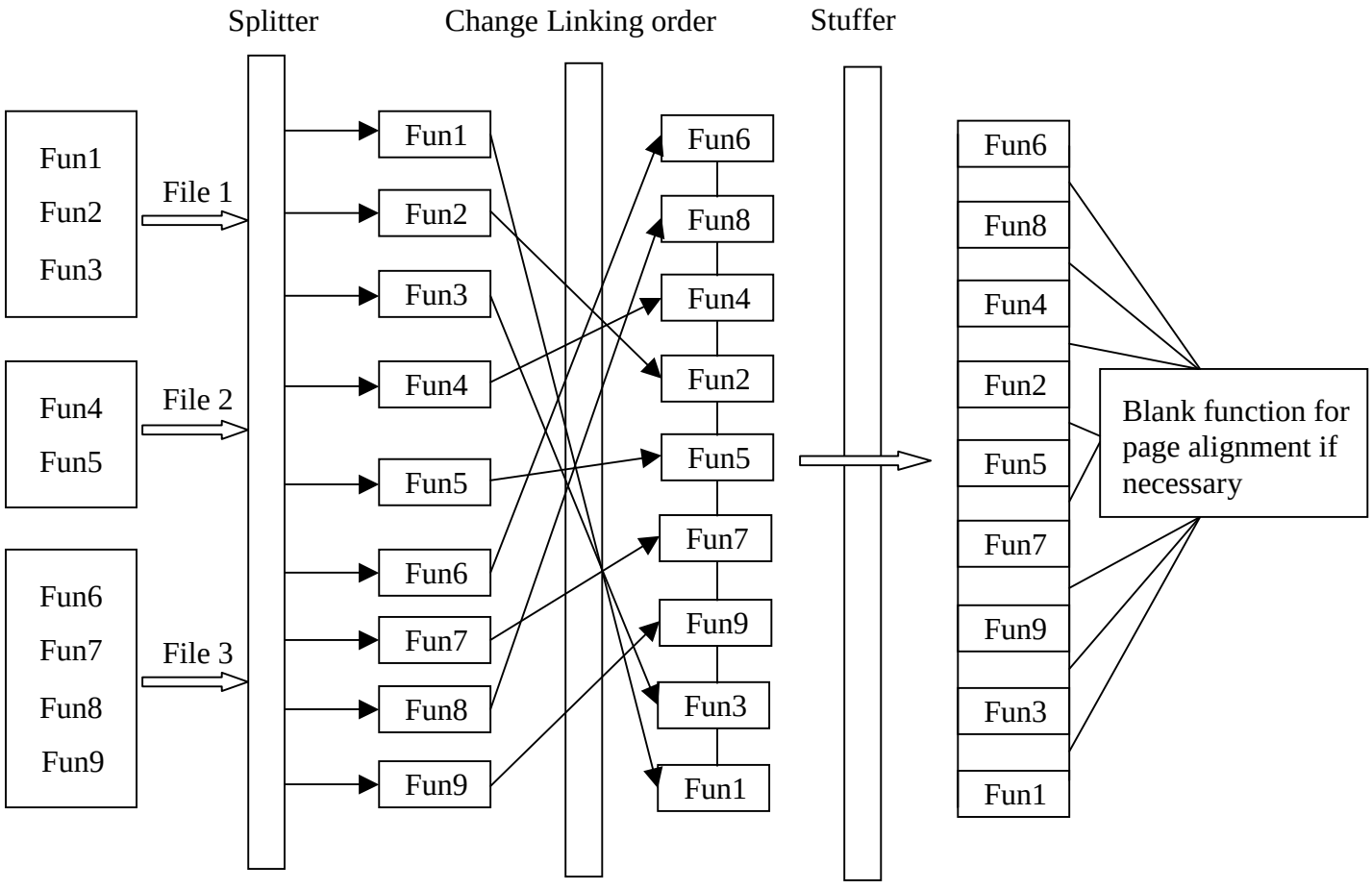

Figure 4. Example of the Slicer working process

3. The System Architecture

3.1 Overview

The Slicer is a C source code tool created by Unix utilities — Lex, Yacc and Perl language. It is composed of three basic parts — the Instrumenter, the Splitter and the Stuffer, which together support the long-running large application's optimization. Figure 5 shows the system architecture of the Slicer. From it, we can see that the Slicer consists of the Instrumenter, the Splitter and the Stuffer. The Instrumenter is in charge of the instrumentation of the program source files and generating the calling patterns. The Splitter handles the splitting work, i.e., it extracts every function from the source files so later they can be linked in any order to realize the rearrangement of the function's memory layout. The data miner analyzes the calling patterns produced by the Instrumenter and generates optimized functions' memory layout. The Stuffer is used to figure out the blank functions' positions and sizes and to generate an optimized executable file according to the data miner's information. In Figure 5, the arrow line originating at the Optimized Executable pane represents the feedback adjustment for the data miner to provide more input to the optimizations.

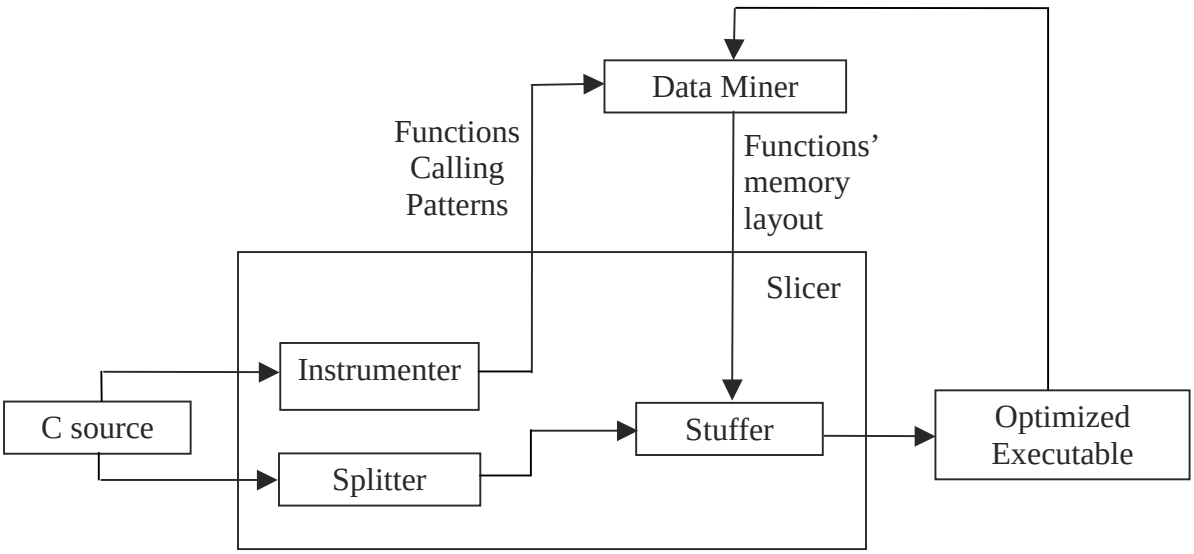

Figure 5. Architecture of the Slicer

3.2 The Instrumenter

Basically, the Instrumenter is created with the Unix utilities Lex and Yacc, but it is actually not a complete C parser, since it only parses out information and positions in which we are interested, such as the start and end of a function, the macro definitions, the include macro definitions, etc. We have four fundamental functions to insert into the target program:

```
INSTR_TRACER_INIT();              // do some initialization work
INSTR_TRACER_ENIT();              // do some clear work before the end
INSTR_TRACER_START( int id );     // inserted at the beginning of each function
INSTR_TRACER_END( int id );       // inserted at the end of each fuction
```

Among them, function INSTR_TRACER_START() and INSTR_TRACER_END() will record the current time, assign the id that is unique to every function, and write the Start/End tag to a log file. Hence, we can get the program calling patterns from the log file. For a multiple process/thread program, we add a sharing policy to avoid two processes from accessing the log file in the same time, i.e., to keep the file's consistency.

There were some challenges to make the Instrumenter work properly. First, some functions have multiple exit/return points, so the Instrumenter have to parse all of them out (e.g. exit(), return(), etc.), and add function INSTR_TRACER_END() there. The records of functions' start points and end points must match in pairs. Second, some functions appear in Macro definitions format. For example,

```
#define my_function( NAME, TYPE ) TYPE NAME( TYPE x ) { return x; }
my_function( function1, int )
my_function( function2, float )
```

The TYPE could be "int", "float", or "char" and the real function name is given by the Macro parameter NAME. If the source file is precompiled with command "gcc –E file.c", then the output is:

```
int     function1 ( int   x ) { return x; }
float   function2 ( float   x ) { return x; }
```

So we actually face two functions to instrument. The Instrumenter solves this kind of problems by pre-compiling the source files first, before applying its own parser. Then all the hidden functions defined in Macro definitions are revealed. Since pre-compiling will

reduce the readability of the source file and make error checking difficult, we use it just for solving this special problem. For other macro-definition problems we meet in the Splitter, we still use the parser techniques to solve them.

3.3 The Splitter

The Splitter is a tool for parsing every function from the source files and saving it to a file named after the function's name. The purpose of the Splitter tool is to group and stuff related functions at the source code level to circumvent the complexity of binary code level optimizations. The Splitter is also created with Lex and Yacc and there are many similarities between the Instrumenter and the Spliter. As a matter of a fact, the Splitter faces and needs to solve most of the same problems, such as tokens and syntax analyses, as the Instrumenter does, so we will not mention them again. Here we just focus on the specific features of the Splitter.

Generally speaking, a file will contain several functions and these functions my exchange some information through global variables. When splitting, we should avoid multiple definitions by declaring the global variables once only in a function file and giving 'extern' reference in other function files. For the static global variables, the Splitter has to remove their static properties, so that they are viewable among different function files, and continue acting as an information exchange method.

For the header files and constant definitions included in each source file, the Splitter collects them together, and saves them to a header file named after the source file name. When splitting, the Splitter will add this new header file to all of the function files, which makes the splitting process clear and brief. For the conditional definitions, such as:

```
#ifdef    HAVE_CONDITION
          function1() { … } // body11
          function2() { … } // body21
#else
          function1() { … } // body12
          function2() { … } // body22
#endif
```

the Spliter will keep this macro definitions structure and split the piece of code as follows:

```
#ifdef      HAVE_CONDITION          #ifdef      HAVE_CONDITION
            function1() { … } // body11                  function2() { … } // body21
#else                               #else
            function1() { … } // body12                  function2() { … } // body22
#endif                              #endif
```

Compared with the pre-compiling, the advantage of doing so is that the split code will still be easily changeable and suitable for the same platforms as before.

3.4 The Stuffer

The Stuffer is written in Perl language, and it is used to get the optimized executable file. To achieve that, the Stuffer needs three inputs: the split functions, the optimized memory layout, and some blank stuffing functions with different sizes for desired page alignment. The first two items can be obtained from the Spliter and the data miner output, but it needs to generate blank functions by itself. The overall workflow of the Stuffer is showed in Figure 6 below.

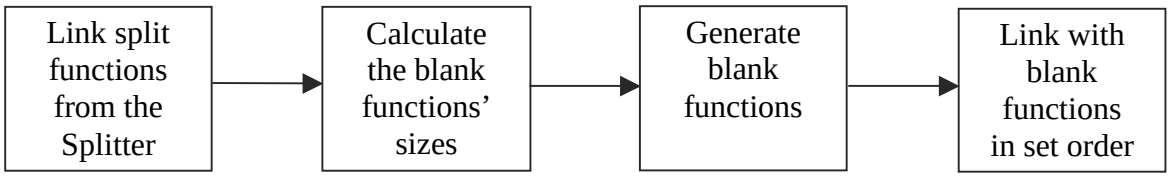

Figure 6. Workflow of the Stuffer

The first task for the Stuffer is to find the blank functions' positions and lengths. Using the sequence from the data miner, the Stuffer compiles and links a temporary executable file in the beginning. Since there may be several thousand functions (e.g., about 2300 functions in Apache HTTP server), the compiling and linking command will be generated by a Perl script automatically. From this temporary executable file, we can see the length of each function in binary format using Unix command "nm –n". For example, below is a piece of output that we get using the "nm" command. In this example, the code piece after the system function "init_dummy" is the beginning of the original code.

```
0804b750 t frame_dummy
0804b770 t init_dummy
0804b778 T __rpi_cs_ngs_lzzhao_stuff_0
0804b778 t gcc2_compiled.
0804c000 T apr_array_push
0804c090 T apr_pstrdup
0804c0d0 T ap_add_node
0804c110 T substring_conf
0804c180 T ap_getword_conf
```

We can figure out the length of the function “apr_array_push” as follows: 0x0804c090 – 0x0804c000 = 0x90 (a Hex number). The function “__rpi_cs_ngs_lzzhao_stuff_0” is actually the Stuffer’s first blank function, that makes the starting point of the function “apr_array_push” page-aligned (i.e. address mod 4096 = 0). From these functions’ lengths, we can easily figure out the blank function position and length according to the sequence set by the data miner, just like Figure 2 shows.

Next, we need to generate the blank functions with suitable binary lengths. According to our experiments on Linux platform, the binary size of functions increases by 4 byte steps, and the minimum size is 12 bytes. According to the length requirement, the Stuffer will generate the functions as follows:

```
void  __rpi_cs_ngs_lzzhao_stuff_N(void) {
      unsigned char a[100];

      a[0] = 0;
      a[1] = 0;     // start
            … …
      a[2] = 0;     // end
}
```

Assuming the number of similar lines between (and including) the “start” and the “end” is N, the final binary length of the function is 12+N*4. With this expression, a Perl function of the Stuffer called “creat_fun()” will automatically generate blank functions according to the corresponding input parameters of length.

At the end of execution, the Stuffer will produce a linking command, which will call ‘gcc’ again to link the split functions as well as the blank functions together to form the final optimized executable file with page alignment.

4. Application to Apache

Apache HTTP server is a famous open-source HTTP server running on Linux and other platforms. It is also a large long-running program. According to different specific services and contents (e.g., server for news publication, web email server, Bulletin Board System, etc.), Apache servers generate various access patterns. All of these properties make it an ideal object for the Slicer to optimize.

4.1 Behaviors Collection

The first step of our optimization process is to collect Apache server's behavior patterns that represent the general access pattern. As we know, the Apache server has an access log file "access.log" to record every access to the server, including such data as IP address, date and time, HTTP method and content requirement. We use this file to replay the historical access actions to our Apache server instrumented by the Instrumenter. We will not run the modified server directly for user services because the instrumented server runs much slowler than the original one. There are about 2300 functions in the Apache package, and several function's calls will result in a disk writing, which depends on the temporary writing buffer size. Furthermore, the log files may span several weeks. Luckily, what we want now is just function's calling sequence, not fast running, so we can replay these log files on a testing Apache server and wait with patience for the calling sequence generation. The sequence content may include: calling time, function identity, start or end. To minimize the size of the sequence record file and the replaying time, we use as few bits as possible to represent the needed content. Finally, the collected patterns are transmitted to the data miner for frequency and sequence analysis.

4.2 Functions' Splitting

The function's splitting process can be done in parallel with the behavior pattern collection. Just as Figure 5 shows, they are not dependent on each other. For Apache HTTP software package, there are about 550 C source files located in 5 directories. The Splitter will create a directory for each C file, which is named after the C file, and put all the functions originally belonging to the file in the directory. For so many files to split and so many directories to create, to apply our parser one by one manually would be

prohibitively expensive. So, we created a Perl script to create and traverse the directory tree. For holding the global definitions and including declarations, the Splitter produce a header file for each original C file and put all such header files in a directory called "head". So each split function file need to include its corresponding header file. Totally, 550 such header files were created.

4.3 Mining Calling Sequence

Next, the data miner module will analyze the calling sequence collected by the Instrumenter, and group the related functions together. The grouping will follow the following basic rules. First, if two or more functions are always called together, then they should be put in the same page or neighbor pages if their size is more than a page size. For example, function A always call function B, and function B always call function C, then we will put A, B and C together in a page. However, because the calling relationship is not a tree structure, it is possible that a popular function is called by many other functions located in several pages. So there is a tradeoff to decide where we should place the popular function. Second, frequently called functions (either related or not) should be placed together as close as possible. This rule helps to keep the popular functions residing in the memory with highest probability. We can see that data mining techniques is very good at doing this kind of analysis, by using sequence mining and statistics.

4.4 Linking and Stuffing

The new memory layout from the data miner specifies the functions' groups from the first page to the last page of the final executables. Generally, the total functions' size of group is a little less than a page size. However, if a function's size if more than the page size, then it will occupy two or more pages and its name will appear in multiple groups. The left space in a page (i.e. page size – group size) is stuffed by a blank function created by the Stuffer. Since the minimum size of a blank function size is 12 bytes, when the left space in a page is less than that, we will just stuff a blank function with size 12 there. Accordingly, in the next page, we will arrange such a group whose size can fit in there. The detailed generating process is described in Section 3.4.

5. Experimental Results Analysis

We have applied the Slicer to the HTTP server software package Apache successfully. We got some notable experimental results by testing the performance difference before and after optimization of the Apache.

5.1 Traffic Generator

In order to test the performance of the optimized Apache server and generate the calling sequence for the data miner training, we created a traffic generator to send data requirements to the HTTP server. We can view the traffic generator as a re-player of the past server actions, that are obtained from the access log file of the Apache HTTP server. By parsing the access log file, the traffic generator could replay all the historical access actions from users on an Apache server. During this replaying process, we can control the access intervals to let the server work at a full-load condition. We collect the access log files for several weeks, and separate them into two parts: the training data and the testing data. The training data is used to feed the instrumented server to get the calling sequence, and the testing data is used for the performance comparison experiments before and after the optimization.

5.2 Results and Analysis

We implemented the experiments on two Linux platforms, a single 1.5GHz AMD CPU with 256MB memory platform and a 4 CPU cluster with 1GB memory. When testing, we send/replay data request (most of them are GET) commands recorded before to the HTTP server as fast as possible trying to let the server work at a full load condition. At the same time, we also recorded the time from the start to the end of the working period. On the first AMD platform, the computer ran only the Apache HTTP server task when doing the experiments. On the second cluster platform, however, we just chose a light load time to implement the experiments since the second platform is a shared cluster. As a result, the running time can be used as a performance scale to evaluate the performance enhancement of our optimization work. We replayed three log files (about 73k, 519k and 3.4M in length) on the randomly linked Apache HTTP server and the optimized server running on the two platforms, respectively. So there are totally

3×2×2=12 different combinations and running times as showed in the Tables 1 and 2. Figures 7 and 8 presented the graphs created from the data in these tables. On average, about 100 bytes long sequence of the log file will generate a data request command.

From the Table 1 data we can see that the percentage of the gained run-time decreases form 23% to 18%, as the length of the feeding log file increases from 73k to 519k. However when the length of the feeding log file changes from 519k to 3.5M, the run-time enhancement stays almost unchanged. From these experiment results, we can see that, in the beginning, not all the functions are called with their average frequency. Those functions with high call frequency may appear even more frequent than their statistical average, so the performance enhancement is unstable in the beginning. But after sufficient run-time, the benefit tends to stabilize unless the access pattern changes greatly. The experimental results on the cluster platform also show similar properties. Comparing the data between Table 1 and Table 2, we can see that the enhancement in performance decreases as the platform's memory becomes larger. For the three different log files, the performance drops from 23% to 10%, from 18% to 3%, and from 18 to 2%. The memory difference between these two platforms can give a good explain for the drops. Our functional memory optimization goal is to reduce the page faults created by calling functions. However, if the memory size for the running application is large enough so that all the functions can be loaded to the memory at the application's execution start, then our optimization technique will have no advantage over the randomly linking method. That is why our optimization shows little benefits on the cluster platform because of its huge memory size (1 Giga Bytes). But in most cases, at least nowadays, it is uncommon to let an application like Apache HTTP server monopolize such powerful hardware resources.

In general, our experiments have demonstrated a considerable performance increase for long-running application Apache HTTP server, as showed in Figures 7 and 8, and we expect better results later with the enhancement of the data mining algorithm and more experiments.

Length of the Log File (Gained Running-time)	Linking Method	Running Time (minutes:seconds)
73,519 (23%)	Randomly	0:26
	Optimized	0:20
519,492 (18%)	Randomly	3:26
	Optimized	2:55
3,451,435 (18%)	Randomly	22:45
	Optimized	18:48

Table 1. Results on a platform with 1.5G AMD CPU and 256MB memory under Linux

Length of the Log File (Gained Running-time)	Linking Method	Running Time (minutes:seconds)
73,519 (10%)	Randomly	0:10
	Optimized	0:09
519,492 (3%)	Randomly	1:02
	Optimized	1:00
3,451,435 (2%)	Randomly	7:21
	Optimized	7:12

Table 2. Results on a cluster with four CPUs and 1GB memory under Linux

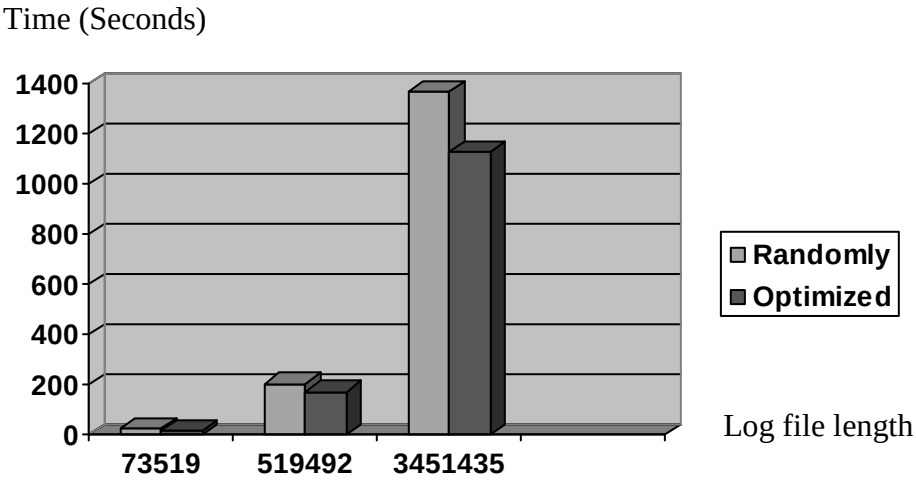

Figure 7. Results on a platform with 1.5G AMD CPU and 256MB memory under Linux

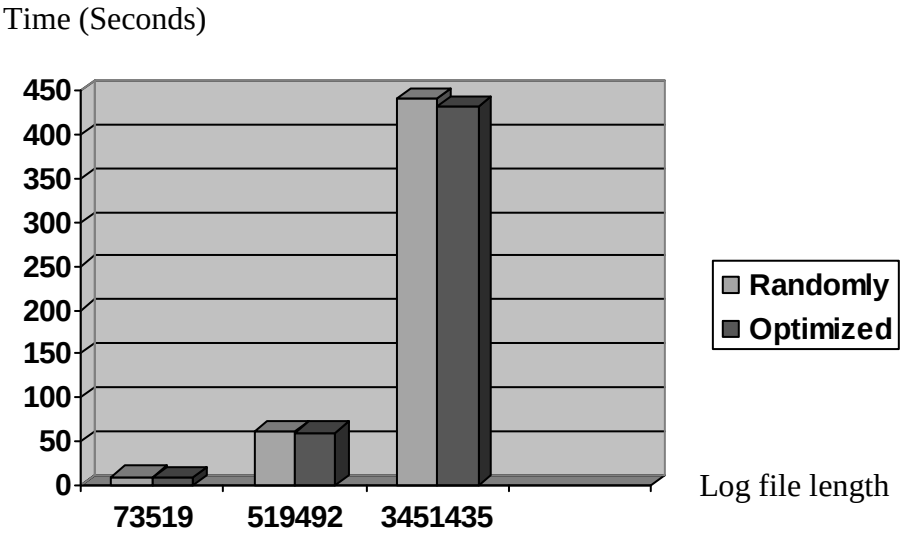

Figure 8. Results on a cluster with four CPUs and 1GM memory under Linux

6. Conclusions and Future work

6.1 Conclusions

This thesis analyzes the relationship between the program code page faults and the memory layout of functions of the program, and presents an optimization method and an implementation tool, the Slicer, based on this relationship analysis. Our assumption for the optimized object is a long time running, large program, which shows semi-stable call patterns of its functions over a considerable time period. Our optimization work is based on three observations. First, if we can put functions that are often called together into the same page, then one page fault is enough to call a group of this kind of functions. Second, frequently called functions should be placed together in one page or several pages, so that a page fault for one of them will load other frequently called functions into the memory, avoiding separate page faults. Such placement also helps to keep those frequently used functions right in the memory. The last observation is that we should let a function span as few pages as possible, so loading it only involves one page fault instead of two if its size is less than a page size.

To prove and realize these ideas, in this thesis, we present a memory optimization tool, the Slicer, to optimize large, long-time running programs. The Slicer is created with Lex, Yacc and Perl, and works at the C source code level. It works in the following basic steps. In the beginning, the Slicer collects the program functions' calling history for predicting the functions future behavior. At the same time, the Slicer splits functions of the optimized program into different files, so it is possible to move them freely when linking them together. Next, we use a data miner to analyze the calling history of functions, and give a better layout of them based on our three important observations said above. In the end, according to the new memory layout of the functions, the Slicer will rebuild a page aligned executable file, which is the output of our optimizations. Here we use the Apache HTTP server as our experimental object, which can satisfy our assumption requirements: a large application, a long-time running program, having similar paging behaviors over a considerable period of time. In addition, the Apache HTTP server is an open source application. After a series experiments on two Linux

platforms, the Slicer optimization demonstrated a considerable performance enhancement meeting our expectations.

6.2 Future Work

After proving the good performance of the optimization, some possible improvements to our optimization method should be considered to make the Slicer more powerful and convenient. The Slicer can only work on C source code now. In the future, we may let the Slicer work on the functional binary code instead on the source code. Though working at binary level may be a little more complex, it will avoid the extra reference information among functions when compiling and linking currently done by the Splitter. Such avoidance will decrease the size of the optimized executable file. Furthermore, the Slicer can also be integrated into a language compiler to make the compiler more intelligent and the optimization process more natural. Then, we do not need to realize the desired page grouping and alignment in the source code level. Instead, we could insert these operations to the code generation phase of the new compiler. In the future, we can also expand this technique to other languages, such as Fortran, C++, etc. Working with the complier would make Slicer the static optimization method.

We could also let the Slicer work dynamically. Currently, we apply the Slicer to the optimized object every so often (e.g. once a week). This optimization process needs to stop the application, do some analysis, and rebuild an application to run it again. Because the overhead imposed by such a process should be much smaller than the performance gained from the optimizations, it cannot be done frequently. However, if we could relocate the functions dynamically in the physical memory as well as virtual memory, then it is not necessary to stop the application for optimizations. Then, we can dynamically monitor the running program, collect performance data, implement the pattern analysis and relocated the functions. The drawback of this method is that it needs a daemon process to do so, which will offset the running program’s performance.

7. Acknowledgements

First, I would like to thank my advisor, Professor Boleslaw Szymanski. He presented me with the good problem space to explore and gave me many detailed directions in the research. Specially, he corrected my paper's reviews word by word, which increased my writing's skill greatly and contributed to the formulation of the thesis. I learned a lot from him and enjoyed it all the while.

During the paper's research, Dr. Mohammed Zaki and Dr. Christopher Carothers, presented many valuable suggestions and technical knowledge, including some edge-cutting scientific papers. In addition, they also helped me to use the Terakava cluster smoothly to implement the necessary experiments.

I would also like to thank my friends for sharing their experience about study and research with me. These experience and information let me find many shortcuts during course study and research.

Finally, and specially, I must thank my wife Jingjing Yu. She gave me lots of support, encouragement and love that helped me out of some pretty rough times. Her considerate companion also made my study life more colorful and enjoyable.

References

[Huang97] J. C. Huang and T. Leng, *Generalized Loop-Unrolling: a Method for Program Speed-Up*, Department of Computer Science, the University of Houston, 1997.

[Cohn96] R. Cohn and P. G. Lowney, *Hot Cold Optimization of Large Windows/NT Applications*, MICRO-29, pp. 80-89, Paris, France, December 1996.

[Franz97] Michael Franz, *Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile-Object Systems*, Mobile Object Systems: Towards the Programmable Internet, 1997

[Cohn97] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin, *Spike: An Optimizer for Alpha/NT Executables*, USENIX Windows NT Workshop, August 11-13, 1997.

[John92] Levine, John R., Tony Mason and Doug Brown, *Lex & Yacc*, O'Reilly & Associates, Inc. Sebastopol, California, 1992.

[Bouq03] Bouqata, Bouchra, Christopher D. Carothers, Mohammed J. Zaki, and Boleslaw K. Szymanski, *Understanding Filesystem Performance for Data Mining Applications*, Proc. 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM:PDS'03) at the Third International SIAM Conference on Data Mining, San Francisco, CA, May 2003.

[Sequ03] Sequeira, Karlton, Mohmamed J. Zaki, Boleslaw Szymanski, and Christopher Carothers, *Improving Spatial Locality using Data Mining*, Proc. 9th International Conference on Knowledge Discovery and Data Mining, P. Domingos, C. Faloutsos, T. Senator, H. Kargupta, L. Getoor (eds.), Washington, DC, August 2003, pp. 649-654.