Phoenix Feedback-Directed Static Compiler Optimization

Shaomei Wu JieJun Xu Nikolay Pavlovich Laptev

Department of Computer Science

University of California, Santa Barbara

{ shaomei, jiejun, nikolay}@cs.ucsb.edu

Abstract

Feedback-Directed Optimization is a common mechanism used for improving program execution based on its run-time behaviors. In this project, we experimented with Phoenix, which is the latest optimization and analysis framework from Microsoft, to learn about how it is used for static FDO. The Phoenix framework provided us a powerful way to extends compiler back-ends by manipulating compilation phases. We have conducted a thorough study on adding different phases into the C++ compiler back-end for both profiling and optimizing application specific programs. In particular, we investigated the effectiveness of 1) profile based Inlining, 2) value profile optimization. Our results showed that Phoenix can be used as an excellent tool for static FDO.

Introduction:

Feedback-Directed Optimization is a well-known method for static compiler optimization.

Before, compiler generates code based on information available only at compile time, and the code generated is often poorly optimized since the information given to compiler is very limited. However if the compiler knows more about the program's run-time behaviours, i.e. how many times a function is called, how many times a specific branch is chosen, and what value is most often to be passed in a function, then the compiler can probably generate much better optimized code. All of above information can be collected as feedback after the program is executed sufficiently enough time. Thus FDO is introduced as a mean for compiler to generate faster code given that run-time information is properly profiled. [3]

In order to collect the run-time information of a program, many profiling tools have been developed. Phoenix is one of such tools developed by Microsoft as the next generation software optimization and analysis framework. Besides basic profiling for program analysis, Phoenix is also capable of generating new codes for program optimization. In addition, it provides a wide range of other functionalities, which are implemented around different level of Intermediate Representations. [2]

The main goal of our project is to learn about how Phoenix interacts and extends C++ compiler back-ends in order to profile and optimize application specific programs. There are several basic optimizations which are already built into the standard C++ compiler. Our goal is to first discover additional optimization opportunities by Phoenix profiling tool, then generate and inject additional optimization code into the original program through manipulating the IR directly

using Phoenix. In particular, we would like to add to the original arsenal of optimizations by including value profiling optimization, and inlining.

The organization of the paper is as follows: Section (2) includes the backgrounds and related work, which gives audiences a general idea of what Phoenix is, and what are the most common ways of FDO; Section (3) describes gives a detail description on our actual implementation of profiling and optimization, i.e. where to add our profiling phase, and what optimization we chose, etc; Section (4) contains the evaluation of our project for both profiling overhead and optimization performance, including the description of our benchmarks, the reason of selecting them and how well our optimization perform on such benchmarks; Section (5) includes our conclusion on this project as well as possible future works. Some potential extension of our work is presented in section (6). At the end, we attach an appendix which contains IR code comparison and a step by step guide on our approach of adding specific Phases / Plugins into Phoenix, this should provide the information necessary for people with very little knowledge on Phoenix to get started and quickly grab a good understanding on Phoenix.

Background and related works

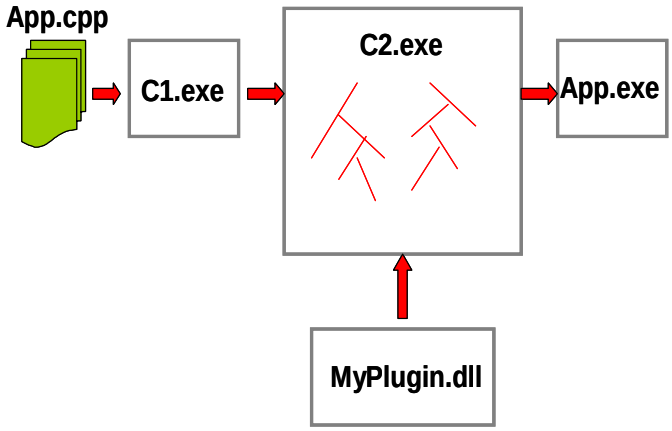
1. Related work in FDO

It is well acknowledged that profile information can be used to overcome the inaccuracy and potentially expand the scope of traditional optimizations. A lot research has been done on Feedback Directed Optimization (FDO). As one of the earliest work on profile-directed code optimization[1] described the design and implementation of a profile-assisted classic static code optimization compiler. In this work, they designed two components of the optimized compiler: execution profiler and profile-based code optimizer. By detecting new optimization opportunities with profile information, they further optimized many classic optimizations such as constant propagation, copy propagation, constant combining, common subexpression elimination, redundant load elimination, redundant store elimination, dead code removal and loop optimizations.

In our project, we selected to implement three kinds of optimization based on profile information: function inlining and variable profiling optimization.

2. Phoenix

As mentioned above, Phoenix is a powerful tool used for software profiling and optimization. One of the most important features Phoenix enabled is the use of Plugins[2].



As shown in the above graph, App.cpp represents the source files, which together make up an application. Cl is the front-end C++ compiler typically comes with Visual Studio. C2 is the back-end compiler, and it has a number of phases. The output from C1 will be process by such phases one after another. Users can much extend the back-end compiler's ability by design specific plugins to insert additional phases to analyze and manipulate the IR code during program compilation. In this way, we can extend the back-end compiler to perform various tasks such as profiling and optimization. And eventually, the 'cl' compiler will produce a new executable code which is generated from a customized compilation phases' list. [2]

Other great features in Phoenix include: (1) Program semantics is well-encapsulated in a large set of APIs: with which Phoenix can provide a very strong instrumentation functionality to enable instrumentation at any Intermediate Representation (IR) level before the executable code is emitted; (2) Different levels of program abstraction from machine-independent high-level to machine dependent low-level [phoenix help file] .We are currently working on HIR level at the timing of two phases: "MIR Lower" in which we can access most semantic information, and "CxxIR Lower" in which Phoenix provides basic blocks view.

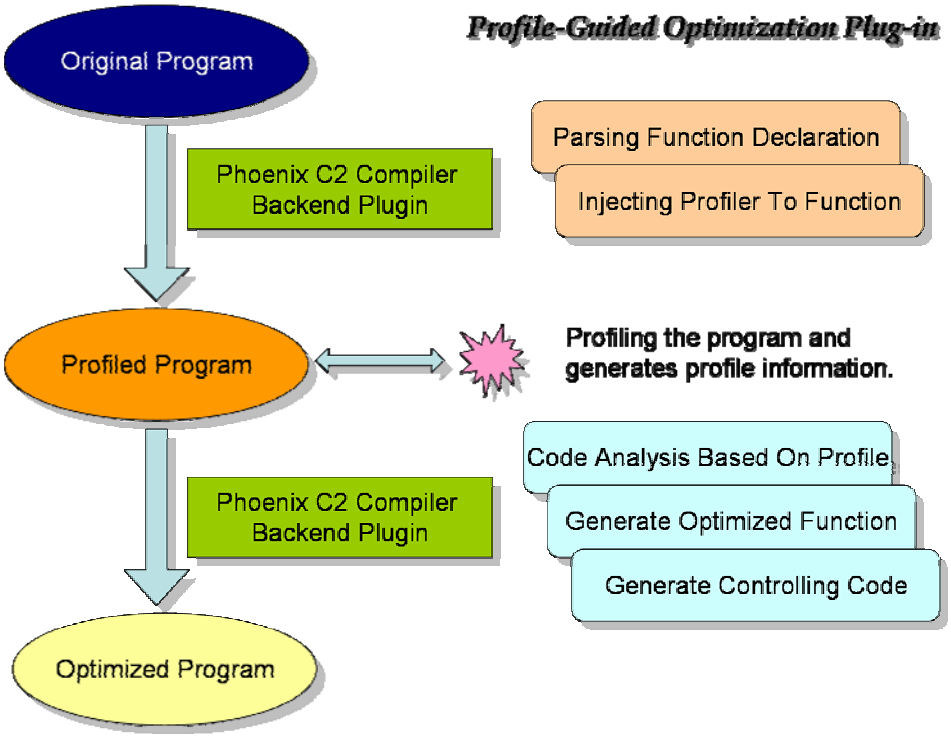
As a new and un-commercial product, Phoenix is not very mature yet. Here are some major comments we have for Phoenix:

- Lack of documentation: the documentation with Phoenix is very incomplete and some are out-of-date;
- Lack of practical development, which means there are no enough project samples to learn from and no many developers to discuss about technical problems;
- Some architecture design program: with instruction manipulation available on only FunctionUnit level, it is hard to operate different functions simultaneously.

Implementation

1. Framework

Here is the system architecture of our profile guided optimization project.



Currently, we analyze the profiling information by ourselves without any scripts or tools, profile analyzer is our future work.

2. Profiler

In our project, profiling of c++ source code is implemented by instrumentation to the executable code of original source. A plugin is created and inserted after "CxxIR Reader" phase to walk through all the functionUnits and insert output instructions inside function bodies. The detailed process is shown step by step in Appendix II.

(1) What to profile:

To do inline optimization and variable prediction, we have to record the hotness of functions and the run-time value of their arguments. We implemented a simple profiling scheme which only output the function name and the argument name and its value when a function is called and executed. Also, we output the size of each function.

(2) How to profile:

We do profiling by instrumentation: be more specific, we inject a "printf" statement at the entrance of each function to output the name of current function, and the name and value of first primary argument (if there is no primary argument, we output null instead).

As a process of recording run-time behavior of the program [1], profiling will also introduce some overhead to the original program. We will show the profiling overhead by comparing the running time of series programs with and without profiler instrumented.

3. Optimizer

An Optimizer is implemented as another Plugin working in a "Profile Guided Optimization" phase after "MIR Lower".

After profiling information is generated and analyzed, we make optimization decisions and save them into a text file, insert a new phrase after MIR Lower phrase to optimize input code.

(1) Value profile optimization:

The value profile optimization is implemented in the execute method of ProfileGuidedOptimization Plugin. We created a new phase called "Profile Guided Optimization" and insert it after "MIR Lower" phase in Phoenix phase chain.

By profiling the value of variables during run-time, we can precisely detect the pattern of certain variables and utilize it to identify opportunities for further static optimizations. In our project, we focused on argument value prediction, i.e., after detecting that an argument (int x) of a function is frequently passed as "0", then we generate a optimized version of this function with preknown value of the argument as constant "0", there are two optimizations we implemented:

- Constant population: we substitute the constant value "0" in expressions in HIR code before x is assigned other value for the first time;
- Branch prediction: after last optimization, we can further simplify conditional branches with all constant values in the condition expression to one unconditional "Goto" instruction, thus eliminate the "Cmp" instruction.

After applying the schemes above, we have two versions of the function – one is optimized for (x=0) and another is not. Phoenix accesses functions only at functionUnit level, it is very hard to create a new FunctionUnit and place it into parent's moduleUnit. Instead, we chose to put these two versions together as two independent blocks in the original function. Also, right after the function's entry point, we insert a conditional branch instruction to check whether current input x is 0 and then jump to proper block. For more idea about how we modify original instructions and implement this optimization, please refer to the variable profile optimization example in Appendix I.

(2) Inline

Inlining is one of the most common compiler optimization techniques. The basic idea is to eliminate the function invocation overheads by "merging" the callee function into the caller function's body. Without profiling information, the decision of whether or not inlining a function is solely based on the compiler's built-in algorithm to estimate the cost and effect of inlining. However this estimation might not be very accurate since the compiler has very limited information available at compilation time besides the function size. With the additional profile information, the compiler can actually see how often a function is called, and whether it is called from a same function. [3]

In our experiment, we have used our profiler to identify all hot edges in a program. By parsing the profile, we could quickly determine the functions which are worth inlining. However, directly implementing our own inlining scheme and injecting in to the IR through Phoenix seems adding too much complexity and it is not worthwhile. We have gone through most part of the Phoenix documentation, and there's no mechanism such as force a function to be inlined on-the-fly. We also noticed that there are quite a few discrepancies between the official documentation and the March 2007 release of Phoenix. In this new release of Phoenix, many of the inlining related methods have been modified. This somewhat support

our assumption that the currently version of Phoenix is immature in dealing with inlining. We have discussed about this problem for quite a while, we thought the cause of such problem is that in the current version of Phoenix, our plugins can not modify the global property of the program and let the compiler iteratively refine the program. Unfortunately inline is such a property that when our plugin discovers to inline one 'functionUnit', it has already processed several previous 'functionUnit's, which might contain calls to this 'functionUnit'. Suppose a 'functionUnit' A calls another one B, and the program execution is upon entering B. It then discover that B should be inlined based on the previous profiles, however A is not longer reachable within the same run due to the constraints of current Phoenix architecture. We believed that the Phoenix research team is going through the process of re-designing the inlining functionalities

Besides probing for solution by ourselves, we have also post the inlining problem on the newly created Phoenix forum, and the only answer we got is to enable inlining through standard command-line options, which is not through the implementation of our plugins. This again support our assumption of the lack of support of inlining in Phoenix.

Although we couldn't directly modify the IR and inject our inlining code, we knew that that could always go back and modify the original source code. Thus, we have implemented a simple Perl script to parse the original source code, and added the keyword 'inline' to each selected function. We believed that this is an easy and efficient way to implement inlining.

```
for (int a0 = 1; a0 <= 1000 + x; a0 ++)
{
    for (int a1 = 1; a1 <= 1000 + x; a1 ++)
    {
            ret += x ;
    }
}
```

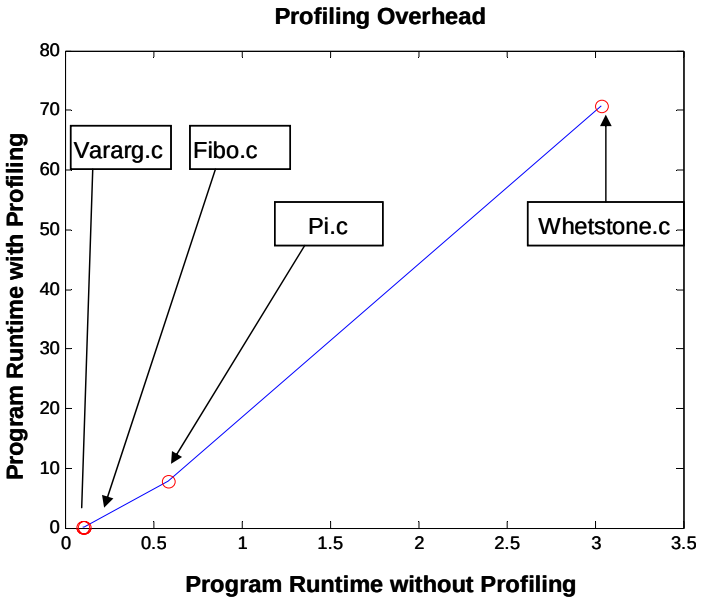
The benchmark we selected for value profiling optimization is included in our attachment. The code snippet shown above is the core part of the benchmark function. Since 'x' is used as part of the conditions in the 'for loop', it cannot be optimized by conventional compiler directly. Due to the lack of information, the compiler couldn't determine the result of the condition with 'x' involved prior to its execution. In our version of optimization, we used the information from the profile to determine which value of the input parameter dominates the function calls, then it enabled us to predict the specific value of that variable, and generate a special optimized version for that function. When the dominated value is passed as function parameter, we called the special function instead of the original version. In the above case, 'x' can be substituted with a constant value, thus we can simplify the compare and the conditional branch instructions so that the execution of the loop is much faster.

Evaluation

1. Profiler sensitivity

As described in Implementation part, our profiler is very sensitive to programs when

there are a lot small functions, i.e. frequent call of small functions is the worst case bringing very high overhead at both profiler runtime and profiling information size. The following figure and table show this phenomenon: with very similar program structures, the five benchmark programs we tested turned out to have almost linear profiling overhead.



	Without Profiling	With Profiling
Vararg.c	0.098310	0.172378
Fibo.c	0.107074	0.200161
Pi.c	0.586308	7.826194
Whetstone.c	3.038422	70.783267
Reinitialize.c	85.770815	> 1800

2. Optimization benchmarks

In our experiment for variable profiling optimization, we turned on the 'O2' – maximize speed switch in both cases. We have called the same function for 500 and 1000 times continuously and keep 80% calls with the same parameter.

	Without optimization	With optimization
Run-time (500 iterations)	1.968 seconds	1.018 seconds
Run-time (1000 iterations)	3.843 seconds	1.921 seconds

Our results showed that our profile-guided optimization achieved about 50% less run time compared to the conventionally optimized version, which is very effective.

The result of inlining optimization is shown as follow:

	Without inlining	With inlining
Run time:	11.43 seconds	5.82 seconds

Our results showed that our optimization achieved more than 50% less run time

compared to the conventionally optimized version.

Summary

Phoenix is very powerful but extremely difficulty to use at current stage. We spent a lot of time to learn and experiment it because of the lack of documentation and samples. As a very complicated system with strict encapsulation, Phoenix should have a least a short description to each class in their APIs. The new tutorial in CGO 2007 really helped us out of the most difficult time of development.

With much more difficulties than we expected, we completed this project successfully with an instrumentation profiler and two optimization schemes implemented in Phoenix. We deployed a series of experiments, collected and designed a set of benchmarks to proved our work by the results shown above.

We think we achieve grade 1 with this work. To develop with Phoenix, a lot of time is devoted. We were also seeking for extra help in Phoenix forum and by emailing Microsoft people who work on Phoenix project. With a large bunch of experiments we have done on Phoenix (because we can't find them stated in documentations…), we learned not only tremendous technical implementation of Phoenix but also a high-level understanding of IR, control flow compiler back-end and optimization. Considering the future work, we also adapted our development onto the new release of Phoenix as soon as it is available even almost all the associated documentation is not up-to-date. The very simple optimization scheme we implemented works very well, which is very rewarding and can be helpful to other programmer on Phoenix in the future.

Future Work

Constrained by the time in such short quarters, there are still some more work left be done in the future:

1. Smarter Profiler (less overhead, more profiling information)
2. Profile analyzer
3. Inlining implemented by Phoenix Plugin

Reference

[1] P. Chang, S. Mahlke, and W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Center for Reliable and High-Performance Computing Report CRHC.91-13*, Univ. of Illinois Urbana-Champaign, Urbana, IL, Apr. 1991.

[2] Microsoft Research, "Phoenix Documentation"

[3] R. Cohn and G Lowuey. "Feedback Directed Optimization in Compaq's Compilation Tools for Alpha," Proc. *Second Workshop on Feedback-Directed Optimization*, held in conjunction with MICRO-33, pp. 3-12, November 1999.

[4] S. Savari and C. Young. "Comparing and Combining Profiles," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with the *32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 50--62, November 1999.

[5] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLANWorkshop on Dynamic and Adaptive Compilation and Optimization*, pages 1¨C11, Boston, MA, January2000.
[6] David W. Wall, Predicting program behavior using real or estimated profiles, Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, p.59-70, June 24-28, 1991, Toronto, Ontario, Canada

Appendix

I. HIR code before and after variable profiling optimization

Source code:

b1.cpp

```
#include <stdio.h>
#include <math.h>

int func(int x)
{
    int ret;
    if (x > 3)
    {
        ret = x - 3;
    }else
    {
        ret = x + 3;
    }

    for (int a0 = 1; a0 <= 1000; a0 ++)
    {
        for (int a1 = 1; a1 <= 1000; a1 ++)
        {
                ret = ret + x;
        }
    }

    return ret;
}

int main()
{
    int i=0, ret = 0;
    for (int i = 1; i <= 2000; i ++)
    {
                if (i >= 600 && i <= 1000)
```

```
                    ret = func(i);
                else
                    ret = func(0);
    }
    printf("ret is %d\n", ret);
    return 0;
}
```

```
Function Unit #1
?func@@YAHH@Z: (references=1)                                   #17
   _x      = ENTERFUNCTION                                      #17
   t276    = COMPARE(GT) _x, 3                                  #20
             CONDITIONALBRANCH(True) t276, $L7, $L6             #20
$L7: (references=1)                                             #20
   t277    = SUBTRACT _x, 3                                     #22
   _ret    = ASSIGN t277                                        #22
             GOTO $L8                                           #23
$L6: (references=1)                                             #23
   t278    = ADD _x, 3                                          #25
   _ret    = ASSIGN t278                                        #25
             GOTO $L8                                           #26
$L8: (references=2)                                             #26
   _a0     = ASSIGN 1                                           #28
             GOTO $L9                                           #28
$L10: (references=1)                                            #28
   _a0     = ADD _a0, 1                                         #28
             GOTO $L9                                           #28
$L9: (references=2)                                             #28
   t280    = COMPARE(LE) _a0, 1000(0x000003e8)                  #28
             CONDITIONALBRANCH(True) t280, $L12, $L11           #28
$L12: (references=1)                                                    #28
   _a1               = ASSIGN 1                                         #30
                       GOTO $L13                                        #30
$L14: (references=1)                                                    #30
   _a1               = ADD _a1, 1                                       #30
                       GOTO $L13                                        #30
$L13: (references=2)                                                    #30
   t282              = COMPARE(LE) _a1, 1000(0x000003e8)                #30
                       CONDITIONALBRANCH(True) t282, $L16, $L15         #30
$L16: (references=1)                                                    #30
   t283              = ADD _ret, _x                                     #32
   _ret              = ASSIGN t283                                      #32
```

```
                    GOTO $L14                                                #33
$L15: (references=1)                                                         #33
                    GOTO $L10                                                #34
$L11: (references=1)                                                         #36
                    RETURN _ret, $L3(T)                                      #36
   t284             = COMPARE(GT) _x, 3                                      #36
                    CONDITIONALBRANCH(True) t284, $L18, $L19                 #36
$L18: (references=1)                                                         #36
   t285             = SUBTRACT _x, 3                                         #36
   _ret             = ASSIGN t285                                            #36
                    GOTO $L20                                                #36
$L19: (references=1)                                                         #36
   t286             = ADD _x, 3                                              #36
   _ret             = ASSIGN t286                                            #36
                    GOTO $L20                                                #36
$L20: (references=2)                                                  #36
   _a0       = ASSIGN 1                                               #36
               GOTO $L22                                              #36
$L21: (references=1)                                                  #36
   _a0       = ADD _a0, 1                                             #36
               GOTO $L22                                              #36
$L22: (references=2)                                                  #36
   t287      = COMPARE(LE) _a0, 1000(0x000003e8)                      #36
               CONDITIONALBRANCH(True) t287, $L23, $L28               #36
$L23: (references=1)                                                         #36
   _a1              = ASSIGN 1                                               #36
                    GOTO $L25                                                #36
$L24: (references=1)                                                         #36
   _a1              = ADD _a1, 1                                             #36
                    GOTO $L25                                                #36
$L25: (references=2)                                                #36
   t288    = COMPARE(LE) _a1, 1000(0x000003e8)                      #36
             CONDITIONALBRANCH(True) t288, $L26, $L27               #36
$L26: (references=1)                                                #36
   t289    = ADD _ret, _x                                           #36
   _ret    = ASSIGN t289                                            #36
             GOTO $L24                                              #36
$L27: (references=1)                                                #36
             GOTO $L21                                              #36
$L28: (references=1)                                                #36
             RETURN _ret, $L3(T)                                    #36
$L5: (references=0)                                                 #37
             UNWIND                                                 #37
$L3: (references=2)                                                 #37
```

```
            EXITFUNCTION                                          #37
$L2: (references=0)                                               #37
            END {*StaticTag}                                      #37



********** Optimized Version of the same function*****************
Function Unit #1
?func@@YAHH@Z: (references=1)                                          #17
   _x               = ENTERFUNCTION                                    #17
   t290             = COMPARE(EQ) _x, 5                                #17
                      CONDITIONALBRANCH(True) t290, $L30, $L29         #17
$L30: (references=1)                                                  #17
                      GOTO $L7                                        #20
   t276             = COMPARE(GT) 5, 3                                 #20
                      CONDITIONALBRANCH(True) t276, $L7, $L6           #20
$L7: (references=2)                                                   #20
   t277             = SUBTRACT 5, 3                                    #22
   _ret             = ASSIGN t277                                      #22
                      GOTO $L8                                        #23
$L6: (references=1)                                                   #23
   t278             = ADD 5, 3                                         #25
   _ret             = ASSIGN t278                                      #25
                      GOTO $L8                                        #26
$L8: (references=2)                                                   #26
   _a0              = ASSIGN 1                                         #28
                      GOTO $L9                                        #28
$L10: (references=1)                                                  #28
   _a0              = ADD _a0, 1                                       #28
                      GOTO $L9                                        #28
$L9: (references=2)                                                   #28
   t280             = COMPARE(LE) _a0, 1000(0x000003e8)                #28
                      CONDITIONALBRANCH(True) t280, $L12, $L11         #28
$L12: (references=1)                                                  #28
   _a1              = ASSIGN 1                                         #30
                      GOTO $L13                                       #30
$L14: (references=1)                                                  #30
   _a1              = ADD _a1, 1                                       #30
                      GOTO $L13                                       #30
$L13: (references=2)                                                  #30
   t282             = COMPARE(LE) _a1, 1000(0x000003e8)                #30
                      CONDITIONALBRANCH(True) t282, $L16, $L15         #30
$L16: (references=1)                                                  #30
   t283             = ADD _ret, 5                                      #32
   _ret             = ASSIGN t283                                      #32
```

```
                    GOTO $L14                                          #33
$L15: (references=1)                                                   #33
                    GOTO $L10                                          #34
$L11: (references=1)                                                   #36
                    RETURN _ret, $L3(T)                                #36
$L29: (references=1)                                                   #36
  t284              = COMPARE(GT) _x, 3                                #36
                    CONDITIONALBRANCH(True) t284, $L18, $L19           #36
$L18: (references=1)                                                   #36
  t285              = SUBTRACT _x, 3                                   #36
  _ret              = ASSIGN t285                                      #36
                    GOTO $L20                                          #36
$L19: (references=1)                                                   #36
  t286              = ADD _x, 3                                        #36
  _ret              = ASSIGN t286                                      #36
                    GOTO $L20                                          #36
$L20: (references=2)                                                   #36
  _a0               = ASSIGN 1                                         #36
                    GOTO $L22                                          #36
$L21: (references=1)                                                   #36
  _a0               = ADD _a0, 1                                       #36
                    GOTO $L22                                          #36
$L22: (references=2)                                                   #36
  t287              = COMPARE(LE) _a0, 1000(0x000003e8)                #36
                    CONDITIONALBRANCH(True) t287, $L23, $L28           #36
$L23: (references=1)                                                   #36
  _a1               = ASSIGN 1                                         #36
                    GOTO $L25                                          #36
$L24: (references=1)                                                   #36
  _a1               = ADD _a1, 1                                       #36
                    GOTO $L25                                          #36
$L25: (references=2)                                                   #36
  t288              = COMPARE(LE) _a1, 1000(0x000003e8)                #36
                    CONDITIONALBRANCH(True) t288, $L26, $L27           #36
$L26: (references=1)                                                   #36
  t289              = ADD _ret, _x                                     #36
  _ret              = ASSIGN t289                                      #36
                    GOTO $L24                                          #36
$L27: (references=1)                                                   #36
                    GOTO $L21                                          #36
$L28: (references=1)                                                   #36
                    RETURN _ret, $L3(T)                                #36
$L5: (references=0)                                                    #37
                    UNWIND                                             #37
```

```
$L3: (references=2)                                                    #37
                    EXITFUNCTION                                       #37
$L2: (references=0)                                                    #37
                    END {*StaticTag}                                   #37
```

II. How to….

1. Run Profiler

(1) Use Phoenix Plugin wizard to create and insert a Plugin working in a new phrase after "CxxIR Reader"; name the Plugin as "VariableProfiling" (Project name and plugin name) (see C2 Walkthrough in Phoenix documents for a detailed guide of this process.)
(2) Replace VariableProfiling.cs generated by Visual Studio 2005 with our VariableProfiling.cs
(3) Compile plugin (F6) in VS2005
(4) Start Phoenix Debug Environment, go to the directory which contains VariableProfiling.dll file.
(5) Copy the benchmarkfile (xxxx.cpp or xxxx.c) into the same directory
(6) Compile without profiler: cl xxxx.cpp
(7) Run the original exe program and see its performance: xxxx
(8) Add profiler: cl xxxx.cpp –d2 plugin:VariableProfiling.dll
(9) Rerun the exe program: xxxx
(10) You will see that profiler is added and profiling info is generated

2. Run Optimizer

(1) Use Phoenix Plugin wizard to create and insert a Plugin working in a new phrase after "MIR Lower"; name the Plugin as "ProfileGuidedOptimization" (Project name and plugin name) (see C2 Walkthrough in Phoenix documents for a detailed guide of this process.)
(2) Replace ProfileGuidedOptimization.cs generated by Visual Studio 2005 with our ProfileGuidedOptimization.cs
(3) Compile plugin (F6) in VS2005
(4) Start Phoenix Debug Environment, go to the directory which contains VariableProfiling.dll file
(5) Make sure you have the optimization guide file under the same directory. In this guide file, each line as this:
funcName=internalFunctionName argName=constantValue
(6) Copy the benchmarkfile (xxxx.cpp or xxxx.c) into the same directory
(7) Compile without profiler: cl xxxx.cpp
(8) Run the original exe program and see its performance: xxxx
(9) Add profiler: cl xxxx.cpp –d2 plugin:ProfilingGuidedOptimization.dll
(9) Rerun the exe program: xxxx
(10)You will see that original program is optimized.