LIGHTWEIGHT, CROSS-PROCEDURE TRACING FOR RUNTIME OPTIMIZATION

BY

ANAND SHUKLA

B-Tech, Indian Institute of Technology, Kanpur, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To my parents.

Acknowledgments

Heartfelt thanks to my advisor Prof Vikram Adve. He has always been very patient and has provided very noteworthy directions in every facet of this work.

I'm also grateful to members of my research group: Chris Lattner, Misha Brukman, Brian Gaeke and John Criswell. They are primary contributors to the LLVM system on which this work is implemented.

Many initial ideas for this work emerged as a result of discussions with David Crowe. I'd like to thank him for his ample patience and very helpful criticisms.

Finally, thanks to Srikanth, Sankalp, and Nirman who made life here at Illinois so liveable.

Table of Contents

List of Figures

List of Abbreviations

FLI First Level of Instrumentation

SLI Second Level of Instrumentation

LLVM Low Level Virtual Machine

FDO Feedback Directed Optimization

JIT Just in Time

CLR Common Language Runtime

PGC Profile Guided Compilation

Chapter 1

Introduction

1.1 Problem Statement

Cross-procedure tracing implies finding frequently executing paths in a program that span one or more procedures. A runtime optimizer detects such paths and optimizes them to improve overall program performance. Even though several instrumentation techniques exist for tracing programs offline, most have high overheads which makes them unsuitable for runtime optimization. In this work we present a lightweight technique to detect cross-procedure traces at runtime.

1.2 Motivation

All modern-day compilers not only convert a program from a high-level representation into a low-level one, they also optimize it for higher performance based on information in the text of the source code. This process is called *static optimization*. It is also possible to make optimizations to the program code while it is running. The process of making performance-improving changes at run time is called *runtime optimization*.

The motivation for runtime optimization lies in the inherent limitations of static optimization. In particular, compiled code is intended to run on generic input, and thus it is optimized for a generic execution. A static compiler that does not know about program execution behavior would fail to capture control-flow bias that may occur at run time, and thus would be unable to identify which sequences of instructions should be optimized most aggressively. In Profile guided compilation (PGC), profiles from execution on sample input are used to guide further optimization. Cohn and Lowney [CL99] show a 17% improvement in SPEC performance with PGC. Nevertheless, PGC has following limitations:

1. For large programs, it is both expensive and difficult to collect extensive profile information that would resemble the actual usage of a program.

2. Aggressive trace optimizations can cause significant increase in code size which may have performance penalty.

3. Trends in software engineering are increasing the obstacles to producing fast code from static compilers. One such trend is the rise of object-oriented programming, which produces modular code that is easier to reuse and maintain. However, some key features of object-oriented programming, such as dynamic dispatch for procedure calls, make static optimizations, particularly inter-procedural optimizations, difficult.

4. Another development that limits the effectiveness of static optimization is the mobile code such as Java bytecode. Such code cannot have any hardware-specific optimizations performed statically, and thus must lose out on significant static optimization opportunities.

Using program behavior during its execution time, and optimizing it online, can provide optimization benefits that can not be exploited by static offline feedback based compilers. Also, a runtime optimizer can optimize programs based on runtime semi-invariant values. However, runtime optimization can be expensive. A runtime optimizer must detect frequently executed program regions, optimize those regions, and modify the existing code to take advantage of optimized regions. Since all these steps must be performed during the execution time of a program, the benefits of optimization will be lost if techniques for detecting hot regions for optimization suffer large overheads.

Trace based runtime optimization provides several desirable features [GBF97, GBF98, YS98]. First, traces provide a large set of instructions that can be assumed to have much simpler control flow, which allows very efficient code scheduling [Fis81]. Second, traces generally constitute a small fraction of code, but form a high percentage of program execution. So any optimization on traces has high returns. Third, traces lead to a very efficient code layout.

In this work we providing a lightweight instrumentation for runtime optimization that detects hot program paths, or traces. We describe our technique for detecting interprocedural paths, and briefly outline how the traces are utilized by a runtime optimizer. We also provide a software trace cache mechanism that allows deploying optimized code at runtime.

1.3 Challenges in Cross-procedure Tracing for Runtime Optimization

Several techniques [BL94, Bal96, Lar99] have been suggested to collect and store trace information that can be used for static FDO. However, such techniques either use expensive instrumentation to collect profiles, or produce profiles that need expensive analysis to recreate hot paths. Both are unacceptable costs for a runtime optimizer.

Some lightweight techniques have been suggested for online detection and optimization of hot java segments [AHR02]. These techniques involve instrumentation of program call graph edges to detect frequently executing functions in a program. They do not yield the actual traces in those functions.

Dynamo [BDB00] and Crusoe [Kla00] partially interpret a program to locate hot paths. However, interpretation can be expensive, and the benefits of optimization must be significant to overcome the overheads of interpretation.

Roar [MTB+01] and rePLay [PL01] present hardware based techniques to detect hot instruction streams. However both methods propose significant additions to hardware, and would not be suitable for existing simpler architectures.

1.4 Contributions of This Work

This work presents a software based mechanism for efficiently tracing programs at run time. We propose a two level instrumentation strategy that can be used for dynamic feedback directed optimization with a very low runtime overhead. We also introduce a simple shift-register based instrumentation that accurately captures paths across function boundaries.

In this work we propose an instrumentation for online FDO that is compiled into a program and discovers hot traces during the program's execution. Since the instrumentation is part of program binary, there is no interpretation overhead. The instrumentation itself is very lightweight, and is designed such that it perturbs rest of the binary code insignificantly. Also, it is easily switched off when no optimization opportunties can be found in a particular execution of a program. The instrumentation yields inter-procedural traces that can be fed into an online optimizer which continually optimizes an executing code.

The implementation and results in this work use the strengths of the LLVM compiler system [LA02]. LLVM is a compiler infrastructure designed for efficient static, link time and run time optimization. In this work we use LLVM system in two significant ways. First, we use the compiler infrastructure to statically analyse and instrument the program

binaries. Second, we store the bytecode along side the program binary, and use the bytecode to efficiently construct control flow graphs at run time. Both these features together allow an efficient implementation of the instrumentation strategy proposed in this work. The implementation techniques discussed here, however, are fairly general, and can be easily applied to any system that would allow efficient analysis of program control flow graph at run time, such as CLR [Mic] and Java JIT [Adl98] systems.

We propose a two level instrumentation strategy. The first level of instrumentation (FLI) is placed during static compilation of a progam binary. This instrumentation is used to identify frequently executing loop regions of the program. The second level of instrumentation (SLI) then locates the frequent paths within the loop regions selected by first level. FLI is a single function call placed on backedges, and is easily converted to a NOP at run time when the instrumentation is no longer desired. SLI is a heavy instrumentation that accurately identifies hot traces, and is invoked only for a short time. When some paths in SLI are determined to be hot, a trace is constructed and passed on to the runtime optimizer for optimization. This thesis describes the FLI, SLI, the subsequent trace generation and deployment of optimized traces. We use FLI and SLI to create instrumentation that can locate hot traces and adapt to program phase behavior. The runtime optimizer is triggered only as a result of instrumentation in FLI and SLI. FLI itself is very lightweight, and when the code with FLI has no optimization opportunities, the FLI is switched off. SLI is done only on segments of code that are likely to yield hot traces. We make the following key contributions:

1. We propose a two level instrumentation strategy that can be compiled into a program, and can yield traces for online FDO without interpretation.

2. We introduce a simple shift-register based instrumentation technique to locate interprocedural traces.

3. Our strategy captures multiple hot paths that begin at the same program point and forms a single aggregate trace. We also present program transformations that increase the number of hot paths in a loop. This technique reduces thrashing in the software trace cache, and allows us to use simple and efficient software trace cache management schemes.

4. We show how a comprehensive and even expensive instrumentation can be used effectively at runtime with little overheads. Our results show that on average the programs have a 2% boost in performance by executing even the unoptimized traces discovered using our strategy.

1.5 Organization of Rest of the Thesis

In chapter 2, we describe related work in the area. In chapter 3, we state our goals and assumptions regarding this work. Chapter 4 describes the first level of instrumentation. As a program instrumented with FLI executes, the FLI identifies the presence of hot loop regions. The hot loop regions are then further instrumented with SLI. The code with SLI for the region is deployed in a SLI cache and stitched to the original program binary. Frequent execution of SLI code leads to either formation of traces, or rejection of both FLI and SLI. Traces are formed from SLI only when there are small number of paths in the loop region that dominate the execution of a loop. When there are no such paths within a loop region, the instrumentation for the region is rejected, and any future execution of the loop occurs out of uninstrumented version of the code for that loop. Chapter 5 talks about the heavier, more comprehensive instrumentation, SLI, that is used to detect hot paths in a loop region. Chapter 6 describes how the SLI code and traces are deployed in SLI cache and software trace cache respectively. Chapter 7 compares the performance of programs instrumented with the techniques described in this work, with uninstrumented programs. chapter 8 describes some of the details in implementation of this work. Chapter 9 concludes the thesis.

Chapter 2

Related Work

Program profiling has been extensively researched. Several schemes exist for profiling programs with minimal instrumentation overhead. Although most schemes work well for feedback directed static compilations, they are inefficient for online feedback directed optimizations.

The *vertex profiling* problem, denoted by *Vprof(cnt)*, is to determine a placement of counters over the program points *cnt* in a CFG such that the frequency of each vertex (or basic block in CFG) in any execution of the program can be deduced solely using the counters placed at points *cnt*. A similar problem is the *edge profiling* problem, denoted by *Eprof(cnt)*. This determines the placement of counters to accurately determine the execution frequency of each edge in the program CFG. Program tracing implies measuring execution frequency of *paths*, or a sequence of basic blocks, within a procedure in a program. Cross-procedure tracing refers to measuring of execution frequency of paths that span across function boundaries.

Knuth [KS73] published efficient algorithms for finding the minimum number of vertex counters necessary and sufficient for vertex profiling, and the minimum number of edge counters for edge profiling. In [BL94], the problem of finding a set of edge counters for vertex profiling, *Vprof(Ecnt)*, is considered. It is shown in [BL94] that for a given CFG, *Vprof(Vcnt)* or *Eprof(Ecnt)*is never better than a solution to *Vprof(Ecnt)*.

Program tracing (or *path profiling*) has its own intricacies. In a dynamic optimization system, edge profile or basic block profile can be useful. However, for formation of actual traces, path frequencies is what is desired. It can be noted that any solution to a vertex profile or edge profile is also a solution to path profile. However, such a solution is not necessarily optimal. In [BL94] it is indicated that solving tracing problem such that minimal set of edges get instrumented is an NP-complete problem. It therefore presents a *Vprof(Ecnt)* solution for intra-procedure tracing. In their work, the instrumentation alone has an overhead of around 17%. Also, it would be expensive to use this kind of instrumentation for a feedback

directed online optimization since the the steps involved in instrumentation are similar to the one needed for retracing the path.

Dynamo [BDB99, BDB00] uses a lightweight speculative trace selection scheme, called as *Most Recently Executed Trace* or MRET. MRET is a dynamic hot trace selection scheme: it uses a combination of interpretation and native binary execution to detect and construct the trace. The Dynamo system keeps a counter for every *start of trace* [BDB99, BDB00] condition when it occurs. When the counter exceeds a pre-set threshold, the system runs in interpretation mode. The interpreter now records the sequence of basic blocks that get subsequently executed in a history buffer. History collection terminates when an *end-of-trace* condition is reached. Since only trace-heads are profiled in Dynamo, there is no information stored about the actual paths of execution. The paths are reconstructed in the manner described, and so they are speculative: the assumption being that the subsequently executed trace is same as the trace being sought.

Our work is similar to dynamo in terms of initial profiling: we also instrument just the backward branches initially. Our scheme, however, differs from MRET in four significant ways. First, it is not speculative since we have exact path information. Our second level instrumenation provides significant information for generating actual hot paths. Second, we do not have to incur interpretation overhead of dynamo. This makes the implementation easy: no complex interpreter need be produced for a dynamic optimization system using our scheme. Also, interpretation is very expensive, and must be avoided as much as possible at runtime. The third factor differentiating our scheme from Dynamo is the code-cache management. Dynamo does not allow removal of individual traces since traces within the code-cache can be connected with each other. So in order to accomodate new traces when the code-cache is full, the whole code cache is flushed. Even though this is able to capture phase-changes, in [HS02] it is shown that a significant fraction of traces in Dynamo are regenerated after flushing. Our scheme allows both addition, as well as removal of individual traces from code-cache. It is possible because of the trace cache policies we use, as described in section 6.1. The fourth important difference from Dynamo is in the way we adapt to phase behavior. We choose an instrumentation strategy that sequentially triggers more comprehensive instrumentation, and then code optimization. Since our initial instrumentation is plugged in and plugged out on demand, we are able to adapt to changing program behavior in much more direct way. In comparison, Dynamo adapts to phase behavior by assuming that a high rate of fragment creation implies a change in program phase. This assumption is partly necessiated because of the organization of fragment cache in Dynamo, which does not allow individual removal of fragments from the fragment cache.

Statistical sampling approaches, such as the DIGITAL Continuous Profiling Infrastruc-

ture (DCPI) [ABD+97] can monitor complete system activity using high frequency sampling. It can reveal frequency of execution of individual instructions and basic blocks, and also pinpoint system bottlenecks by use of processor performance counters. DCPI periodically samples the program counter (PC) on each processor, associates each sample with its corresponding executable image, and saves the samples on disk in compact profiles. The profiling system uses only about 1-3% of the CPU and modest amount of memory and disks. The low overhead is because of sampling period: the periodic interrupts occur approximately every 64K instructions executed in the CPU. Even though the information provided by statistical sampling can be fairly accurate, there is inherent complexity in regenerating the hot-path information at run time since path generation needs extensive analysis on collected data. In particular, no efficient schemes exist for efficiently generating hot paths or hot regions using dynamic feedback of sampling data.

Arnold et al proposed a lightweight mechanism for online detection and optimization of hot java segments [AHR02, AFG+00]. Their technique instruments edges in a call graph to detect hot java procedures. It does not suggest a mechanism for finding hot interprocedural paths.

ROAR [MTB+01] uses a hardware implementation for detecting *hot spots* during program execution. Use of hardware keeps the profiling overhead low. Another hardware based system is rePLay [PL01]. In rePLay, a hardware based sequencer is used to collect retiring instructions to form fragments. The Trace Cache is another hardware based mechanism that aims to capture hot traces [RJSS97, FPP98]. The hardware based schemes propose extensions to hardware, whereas a software based scheme like ours can work on existing simpler hardware systems.

Chapter 3

Assumptions and Goals

3.1 Definitions

Trace Traditionally, a trace is defined as a sequence of basic blocks with a single entry and multiple exits. However, for this work, we define a trace as a set of paths each of which have a single common entry point and multiple exits.

Backedge We define a backedge as an edge going from a node to its ancestor in some DFS spanning tree of the given graph.

Loop region For any backward edge $u \rightarrow v$, all nodes that lie on some acyclic path starting at u and ending at v are considered as forming the loop region for the edge $u \rightarrow v$.

3.2 Goals

We address the following goals in this work:

1. We focus on capturing hot paths in loop intensive programs. We generate traces that can cross multiple procedure boundaries, but form a part of some loop. We do not consider traces that might be formed because of mutually recursive functions. In particular, if a frequently executing program code is not a part of some loop, it can not be captured by our instrumentation strategy. We make following assumptions in the design of our two level instrumentation:

2. When there is more than one hot path in a loop region, we generate a trace combining all hot paths. We combine the hot paths such they have a common entry point. Such a combination of two hot paths is shown in figure 5.13(B).

3. We allow individual addition and removal of traces in the trace cache. We make it possible by not allowing traces to be linked to each other within the trace cache, and generating traces as an aggregate of multiple hot paths that have a common entry.

4. A hot path in a loop may make several function calls. For the purpose of interprocedural tracing, we only focus on paths through callees which do not contain any loops.

3.3 Assumptions

For efficient implementation of this work, following assumptions are made:

1. Instrumentation can be expensive as long as it is executed very infrequently.

2. A program CFG can be efficiently created at runtime.

3. The program binary can be modified at runtime without a large performance penalty.

The first assumption implies that the total cost incurred by any instrumentation can be kept low by ensuring that the instrumentation does not execute for long. In our framework SLI is expensive, but stays for a very short time. Also, FLI is removed after a fixed number of executions.

SLI formation needs analysis of program control flow graph (CFG). However, CFG formation using machine code can be difficult at runtime, and we avoid this cost by using the LLVM bytecode for the program to generate the CFG. For this purpose, the code generator stores the program bytecode along with the program binary during the code generation process. Systems such as CLR [Mic] and Java JIT compilers [Adl98] create CFGs at runtime using the bytecode representation in a simiar fashion.

The third assumption can be a practical constraint for many architectures. Code modification and subsequent instruction cache flushes may have several undesirable effects on the instruction cache performance. For the purpose of this work, however, we assume that such penalties are low. Some other practical constraints for runtime code modification are discussed in [Smi00].

The code originally created by the static compiler with FLI is what we call as the original code. We modify the original code in only two ways. First, an optimized trace, or a SLI code, is linked to the original code by placing a branch from the original code to the optimized trace, or to the SLI code respectively. To delink the trace, or the SLI, we simply remove the branch and reinstate the code that existed in place of the branch. The second way we modify the original code is when we want to remove the FLI: we simply convert FLI into a NOP.

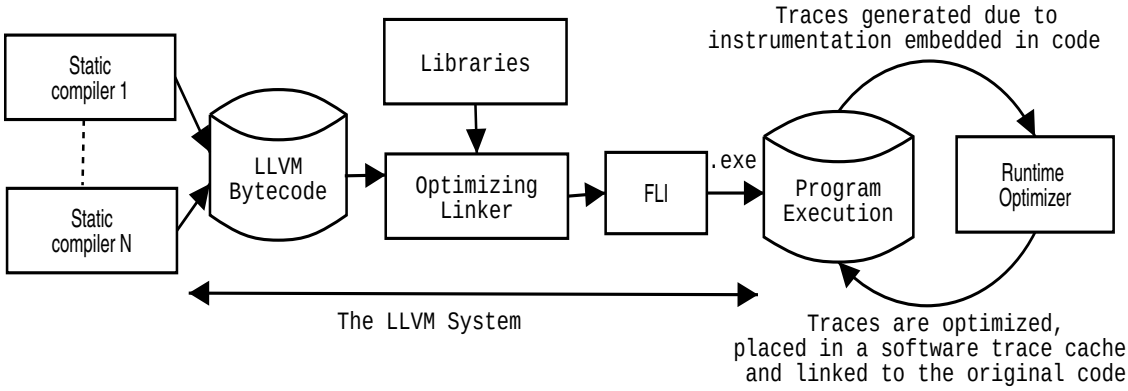

Figure 3.1: The high level organization of a feeback based runtime optimizer

3.4 High-level Organization of an Online Feedback Directed Optimizer

Figure 3.1 shows a high level architecture of a runtime optimization system. This system uses the LLVM compiler infrastructure. A set of source files are compiled into the low level LLVM Bytecode. The LLVM Bytecode uses SSA form and has rich type and dataflow information. It is similar to Java or CLR, but is designed such that aggressive inter-procedural and intra-procedural optimizations can be performed during link time and also runtime. For purpose of runtime optimization, the LLVM based optimizer would typically delegate as much computation to the static compiler as possible. This synergy between the static compiler and runtime optimizer can significantly reduce the overhead of online optimization.

As shown in the diagram, we perform the initial instrumentation just after the program has been linked. The static compiler uses control flow graphs to detect back edges which are best suited for initial instrumentation, as described in section 4.1. In theory, this instrumentation can also be done online. However, doing it offline allows static analysis that can reduce the set of back edges that should be instrumented and would likeley yield hot traces. Morover, static compiler can perform code transformations that would further reduce overall instrumentation overhead. One such transformation is described in section 6.3.

At runtime, the instrumentation embedded in the program invokes the optimizer whenever hot traces are found. The FLI, which was inserted statically, triggers a more comprehensive SLI, which in turn detects and generates hot traces.

Chapter 4

The First Level of Instrumentation

Since the FLI is compiled into the program binary, its design goal is to detect hot loops, but have minimal effect on the original code when no optimization opportunities exist. This implies that FLI should be such that it can be easily turned off, and must have very small footprint so that it does not significantly increase the program code size. Also, FLI must not cause any undesirable effects such as register pressure on rest of the code. In this chapter we describe how the FLI is done, and how its over head can be reduced in two ways: by reducing the program points where it is placed, and by executing the FLI for a short duration.

4.1 Placement of FLI

The FLI is a function call placed at the backedge of loops. The call is to a function called llvm_first_trigger(). This function counts the number of executions of backward branches. Whenever a backward branch executes beyond a threshold, llvm_first_trigger() locates the loop region corresponding to the branch and performs SLI on the selected region.

The algorithm for FLI insertion is shown in figure 4.3. We place FLI such that there is a unique call instruction to first_level_trigger() for every backward branch. Also, we make sure that the function is called only when the branch is taken. This is done by converting every conditional backward branch into an unconditional backward branch, and placing the call just before the unconditional branch, as described in the algorithm in figure 4.3. Figure 4.1 shows FLI for a region of program. The dots indicate the instrumentation which has been placed on unconditional backward branches.

Figure 4.4 shows the pseudo code for the llvm_first_trigger() function. The function has two key aspects. First, it does not take any arguments and does not return any value. Secondly, the function is treated in a special way at the call site. The static compiler does not generate any caller saving code for a call to this function. This function ensures that

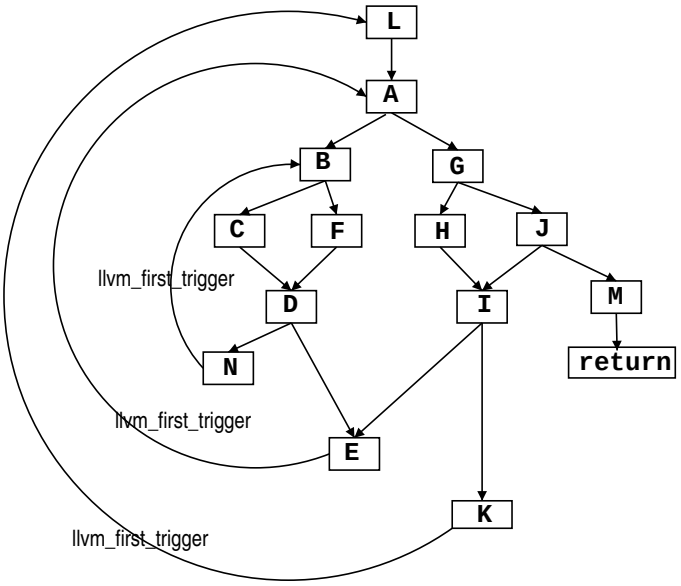

Figure 4.1: The first level instrumentation. This instrumentation is a call to llvm_first_trigger function.

```
GET_BACK_EDGES(Node n, Color C, BackEdges be)
 1  Color[n] ← SEEN
 2  for all successors v of n
 3  do if Color[v] = FRESH
 4        then GET_BACK_EDGES(n, C, be)
 5        else  if Color[v] = SEEN
 6                 then be ← be ∪ (n, v)
 7  Color[n] ← COMPLETED
```

Figure 4.2: Algorithm to get backedges in a CFG

the values used by the caller are not modified. Inside the function, the return address of the caller is used to identify a unique backward branch associated with the FLI. The function uses a hashtable to keep execution counts for all backward branches in the program that have a FLI. When any such branch executes beyond a threshold, the loop region between the branch target and the branch itself is considered to be potential site for hot traces. The region is then modified with SLI to locate hot traces.

```
INSERT_FLI(Graph g)
 1  Set backEdges ← DFS backedges of graph g
 2  for every backedge (u, v) ∈ backEdges
 3  do for every backedge (x, y) ∈ backEdges
 4      do if u properly dominates x
 5            then backEdges ← backEdges \ (x, y)
 6  for every edge (u, v) ∈ backEdges
 7  do create a new graph node t
 8      change the branch u → v to u → t
 9      insert function call to llvm_first_trigger in t
10      insert unconditional branch t → v in t
```

Figure 4.3: Algorithm for performing first level of instrumentation

4.2 Reducing the Overhead of Initial Instrumentation

As is evident from the description, the llvm_first_trigger() is an expensive function call. We take the following two steps to reduce the overhead of the function call:

1. As mentioned in the section 3, the cost of instrumentation can be controlled by limiting the execution of instrumentation for a short duration. The use of a function call for instrumentation allows us to reduce instrumentation to a single instruction. Thus, in order to switch off the instrumentation, it is merely converted into a NOP.

2. We do not instrument every backedge in the program. Several techniques can be used to limit the set of backedges that get instrumented initially. One such technique is shown in the algorithm in figure 4.3. This algorithm uses dominator information to reduce the set of backedges which are instrumented with FLI. One code transformation technique can further reduce the number of backedges in a program is discussed in section 6.3.

In order to limit the execution of the FLI function call, we ensure that for any call site, the function call *does not execute* beyond a fixed number of times. Whenever it exceeds a threshold, it is replaced by a NOP.

Figure 4.5 shows how dominator information is used to limit the backward branches where FLI is placed. For every backward branch that is instrumented with FLI, all other backward branches that are dominated by the branch are not instrumented. As an example, among a set of nested loops, only the inner loop need be instrumented. In figure 4.5A, $A1$

```
global HashTable loop_count;
LLVM_FIRST_TRIGGER()
 1  save caller values
 2  ret ← return address of caller
 3  addr ← address of branch associated with ret
 4  target ← target of the branch at addr
 5  loop_count[ret] ← loop_count[ret] + 1
 6  if loop_count[ret] ≥ FLI_THRESHOLD
 7    then
 8         loop_count[ret] ← 0
 9         convert FLI at ret to NOP
10         root ← basicblock at target
11         BB ← basic block containg addr
12         CFG ← get CFG for the procedure containing root
13         doSLI(root, BB, CFG)
14  restore caller values
15  return
```

Figure 4.4: The llvm_first_trigger function which is called by FLI

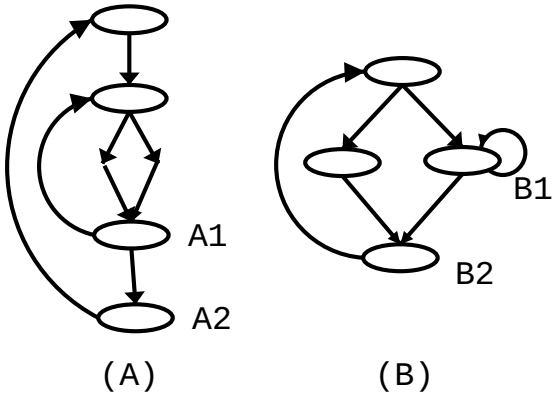

Figure 4.5: Reducing the locations where FLI is placed.

dominates $A2$, so FLI needs to be placed only for backward branch at $A1$. On the other hand, in figure 4.5B, $B1$ does not dominate $B2$, and so both $B1$ and $B2$ are instrumented with FLI.

Chapter 5

The Second Level of Instrumentation

The FLI locates the region of loop that must be further instrumented with SLI. The SLI is a comprehensive instrumentation that sets out to locate the frequency of execution of interprocedural paths in a loop region. When SLI is invoked, the FLI instrumentation form the original code is removed. The generation of SLI entails the following steps:

1. Locate the set of nodes that should be instrumented with more comprehensive instrumentation to identify hot paths. Two sets of nodes, called simple nodes and exit nodes, are identified for instrumentation.

2. Instrument the simple and exit nodes. The simple nodes are instrumented to record path information in the loop. The exit nodes are instrumented to count exits from the loop.

3. When the SLI loop executes beyond a preset threshold, the loop region either yields a trace, or yields no hot paths and is rejected.

5.1 Locating Simple Nodes and Exit Nodes

Every loop region has a unique *root* and a unique basic block (BB) associated with the loop region. Recall that every FLI is associated with a unique unconditional backward branch. The BB is the basic block containing the unconditional branch associated with a FLI that triggered the SLI. The root is the basic block which is the target of the backward branch. Every loop region consists of two kinds of paths. The first kind of path, called simple path (SP) starts at the root, and reaches the BB without traversing any backedge. The second kind of paths are those that are not SP. A non SP path starts at root and either traverses a back edge to reach BB, or does not reach BB at all. Figure 5.1 shows such a loop region.

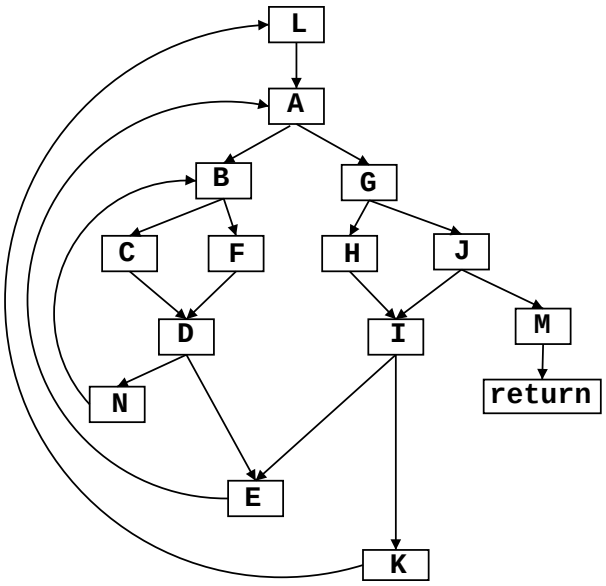

Figure 5.1: A loop region for second level instrumentation.

In the figure, assume that the backward branch in node E is being executed frequently and exceeds its threshold. The loop region for the branch at E has node A as its root, and node E as the BB. The paths ABCDE, ABFDE, AGHIE and AGJIE start from A and lead to E without going through a cycle and are simple paths. The path ABFDNBDFE and AGHIKLABFDE are not simple paths because they traverse the back edges NB and KL respectively. The path AGJM is also not a simple path because it does not reach E.

We form traces only for SPs. For instance, in the figure 5.1, the loop region corresponding to root B and BB N, which is the inner loop, might execute more frequently than the region of root A and BB E. In such a case, we do not want to produce a trace ABFDE if BFD executes more frequently than ABFDE. Instead, in this case, the trace BFDN must be generated for the inner loop because of frequent execution of BB at D.

We first locate all SPs, and instrument every conditional branch on the SPs to record the direction of branch. SPs are located as follows.

1. Starting at the root, do a DFS to locate all nodes that are reachable from the root without traversing any backedge. Call them *reachable nodes*. The algorithm for finding reachable nodes is shown in figure 5.3. It is a DFS based traversal of the graph starting at the root, but avoids traversing any backedges. The backedges are discovered using the algorithm described in figure 4.2.

2. Among the *reachable nodes*, locate the set of nodes that are also ancestors of the BB. These set of nodes are called simple nodes (SN). Figure 5.4 shows the algorithm to find

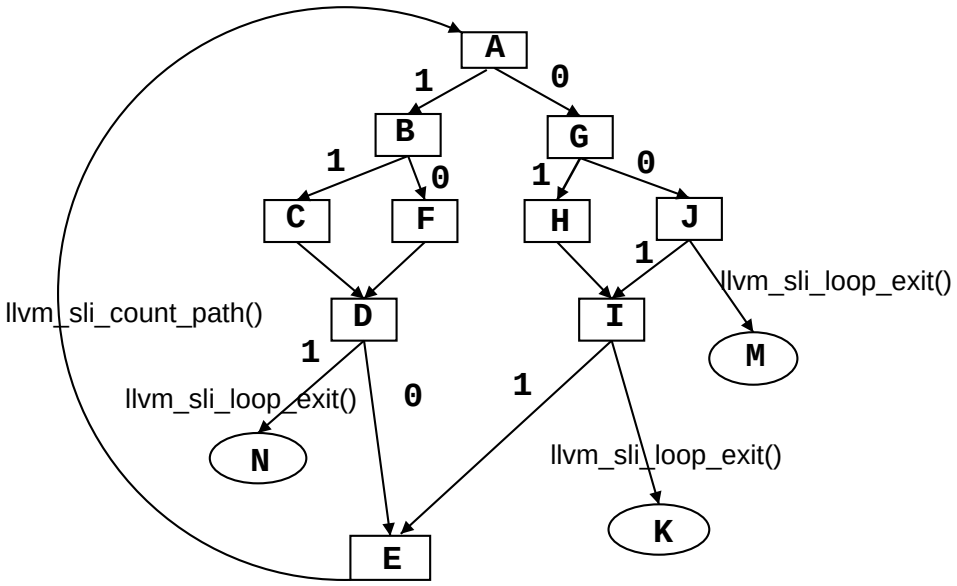

Figure 5.2: Placing instrumentation for a loop region.

```
REACHABLE_NODES(Node N, Node end, Color C, Set backEdges, Set reachable)
1  C[N] = SEEN
2  reachable ← reachable ∪ {N}
3  if N ≠ end
4     then for all successors v of N
5          do if (v, N) ∉ backEdges
6               then if C[v] = FRESH
7                    then
8                     REACHABLE_NODES(v, end, C, backEdges, reachable)
9  C[N] ← COMPLETED
```

Figure 5.3: Algorithm to get reachable node form the Root of a loop region

simple nodes. The algorithm performs a reverse DFS starting at the BB. However, this time, it only traverses the nodes that are *reachable nodes*.

3. For every node in SN, find its successors that do not lie in SN. These successors that do not lie in SN are designated as exit nodes. The algorithm for finding exit nodes is shown in figure 5.5.

4. All paths that start at the root, and reach the BB using only simple nodes are the simple paths (SPs).

```
GET_SIMPLE_NODES(Node N, Node Root, Color C, Set reachable, Set SN)
 1  C[N] = SEEN
 2  SN ← SN ∪ {N}
 3  if N ≠ Root
 4     then for all predecessors u of N
 5          do if u ∈ reachable
 6                then if C[v] = FRESH
 7                        then GET_SIMPLE_NODES(u, Root, C, reachable, SN)
 8  C[N] ← COMPLETED
```

Figure 5.4: Algorithm to get Simple Nodes

```
GET_EXIT_NODES(Set SN, Set exitNodes)
 1  for all nodes N ∈ SN
 2  do for all sucessors v of N
 3       do if u ∉ SN
 4             then exitNodes ← exitNodes ∪ {v}
```

Figure 5.5: Algorithm to get exitnodes of a loop region

In the example in figure 5.1, the nodes N, K and M are exit nodes for the loop region with root A and BB E. These nodes are shown in oval in the figure 5.2. In the example, we find that the set of nodes {A, B, C, D, N, E, G, H, I, K, J, M, return} are set of nodes reachable from A without traversing any backedge. The node L, on the other hand, is not reachable from A without traversing the backedge KL. Among the nodes in the set, {A, B, C, D, E, G, H, I, J} are also the ancestors of E, and form the set SN. In the set SN, the sucessor node K of I is not in SN. Also, the successors N and M, of nodes D and J, are also not in SN. So N, M and K form the exit nodes. In the figure 5.2, the exit nodes are shown in ovals.

5.2 The Instrumentation of SLI Loop Region

The instrumentation for finding path traversed in a loop region uses a integer called the path register. Whenever a conditional branch is taken, a 1 is shifted in the path register value

```
INSERT_SLI(Node Root, Node BB, CFG cfg)
 1  Set backEdges, reachAble, SN, exitNodes
 2  Color C1, C2, C3
 3  for all nodes v ∈ cfg
 4  do C1[v] ← FRESH
 5     C2[v] ← FRESH
 6     C3[v] ← FRESH
 7  GET_BACK_EDGES(cfg.root, C1, backEdges)
 8  GET_REACHABLE_NODES(Root, BB, C1, backEdges, reachAble)
 9  GET_SIMPLE_NODES(BB, Root, C2, reachAble, SN)
10  GET_EXIT_NODES(SN, exitNodes)
11  for every node N ∈ SN, N ≠ BB
12  do for every successor v of N
13     do if v ∈ SN
14           then if branch N → v isconditional
15                   then if if N → v istakenbranch
16                           then insert taken code on edge N → v
17                           else  insert not − taken code on edge N → v
18        if v ∈ exitNodes
19           then insert llvm_sli_loop_exit() on edge N → v
20  insert llvm_sli_count_path() in BB
21  insert the generated SLI code in Tracecache
```

Figure 5.6: Algorithm to form SLI for a loop region defined by a Root and a backward branch at the node BB

from the right. Figure 5.6 shows the algorithm for instrumenting a loop region with SLI. Whenever a conditional branch is not taken, a 0 is shifted. The register is initialized with a 1, so the first 1 in the register indicates the beginning of a path. For instance, consider the value 1100 in the path register for the example in figure 5.2. This indicates that starting at the root A, the first conditional branch was taken, the second conditional branch on the path was not taken, and the third conditional branch on the path is also not taken. This leads to the path ABFDE. The path register can be implemented using a register, or a memory location. For experimental evaluation of this work, we use a machine register in SparcV9 architecture to implement the path register. The figure 5.9 shows the code that is inserted on the taken and not taken conditional directions of a conditional branch.

At the BB for the loop region, a call to a function llvm_sli_count_path() is made. At this point, the path register contains the exact SP traversed from the root to the BB. The pseudo

$$pathRegister = pathRegister \ll 1$$
$$pathRegister = pathRegister \vee 1$$

Figure 5.7: Taken Path

$$pathRegister = pathRegister \ll 1$$

Figure 5.8: Not taken path

Figure 5.9: Code for taken and not taken path

code for the function llvm_sli_count_path() is shown in figure 5.10. This function uses the path register value, as well as the address of the return address to the caller to index a hash table and record the occurrence of each path.

On every exit node in the loop region, a call to llvm_sli_exit_count() is made that keeps count of exits from the loop. The code for this function is shown in figure 5.11. Thus for every loop region, three sets of values are maintained by the instrumentation system: the count of occurrence of each SP in the loop region, the total iterations of the loop region, and the count of exits from the loop region. The next section descibes how a set of hot paths is chosen using this set of information.

The code for a loop region with SLI is placed in a SLI cache, and is linked to the original code through a branch. This branch is placed at the root of the loop region in the original code, and it branches to the top of SLI region placed in the SLI cache. All exits from the SLI always go back to the original code. The organization of SLI cache is further discussed in section 6.1.

5.3 Forming Traces Using SLI

For every SLI, we maintain the total number of executions of all SPs, and also the total number of exits from the SLI loop region. Figure 5.12 describes how a set of paths is chosen to form a trace. A set of SPs in the loop region are termed as hot *only* if

1. the total number of exits from the loop region is a small percentage (M) of the total number of executions of BB, and

```
global HashTable sli_hash;
global HashTable sli_iteration_count;
LLVM_SLI_COUNT_PATH(path_register)
 1  ret ← return address of caller
 2  target ← taget of branch associated with ret
 3  sli_hash[target, path_register] ← sli_hash[target, ret] + 1
 4  sli_iteration_count[target] ← sli_iteration_count[target] + 1
 5  if sli_iteration_count[target] > SLI_THRESHOLD
 6     then generatePaths(sli_hash, sli_iteration_count, target)
 7          remove entry with key {target, path_register} from sli_hash
 8          remove entry with key {target} from sli_iteration_count
```

Figure 5.10: Pseduo code for llvm_sli_count_path() that is called from SLI code

```
global HashTable sli_exit_count;
LLVM_SLI_LOOP_EXIT()
 1  ret ← return address of caller
 2  target ← taget of branch associated with ret
 3  sli_exit_count[target] ← sli_exit_count[target] + 1
 4  if sli_exit_count[target] > SLI__EXIT_THRESHOLD
 5     then remove entry with key {target} from sli_exit_count
 6          delink this sli from the original code
```

Figure 5.11: Pseduo code for llvm_sli_loop_exit() that is called from SLI code

2. a small set of j or less paths in the loop region together account for a high percentage of the times the BB executed in the SLI.

The algorithm uses two preset values: j, which is the maximum number of allowed paths for trace formation and M, the minimum percentage of execution the set of chosen paths should together have. The algorithm locates the set of paths for the loop region, and sorts them by their execution count in decreasing order. It then tries to find the set of i ($i < j$) paths that would together form the bulk of loop iteration count. If such paths are found, a trace is created for these paths as described below. If there are no such paths, it is assumed that the loop region has no clear hot paths, and so there would be no gain in creating traces

```
global HashTable sli_hash;
global HashTable sli_iteration_count;
GENERATE_TRACES(target)
 1  Let P be the set of paths for target in sli_hash
 2  Let k ← |P|
 3  Let C ← sli_iteration_count[target]
 4  Let j ← maximum number of allowed paths for trace formation
 5  Let M ← minimum percentage of execution required for trace formation
 6  Let p_1, p_2, ..., p_k be the execution counts of
 7     k paths s.t. p_1 ≥ p_2 ≥ ... ≥ p_k
 8  for 1 ≤ i ≤ j
 9  do
10       if (Σ_{m=1}^{i} p_m) / C ≥ M
11          then break
12  if i = j
13     then delinkSLI
14     else  generate trace using paths corresponding to p_1, ... p_i
```

Figure 5.12: Algorithm to choose the set of hot paths from SLI

out of the region, and so the loop region is discarded.

Recall that a trace can consist of multiple paths which have a common entry. We generate a single trace for the set of i hot paths as determined by the above algorithm. As an example, figure 5.13 shows the process of forming a trace from two hot paths. In the example from the SLI of loop region in figure 5.2, assume the path AGHIE executes 65% of the total executions of E, and the path AGJIE executes 30% of the total executions of E. In this case, both the paths are hot. We form a combined trace out of the two paths and deploy them in the software trace cache as in figure 5.13B. In the figure, AGHIE is the primary path, and JIE is the secondary path that is linked to G of primary path. The portion IE of the path is tail duplicated. Also notice that the exits from the generated trace go back to the original code. Instead of forming a trace out of a single path, we combine multiple paths to better capture the hot program paths. This also leads to a simpler trace cache design, as further discussed in section 6.1.

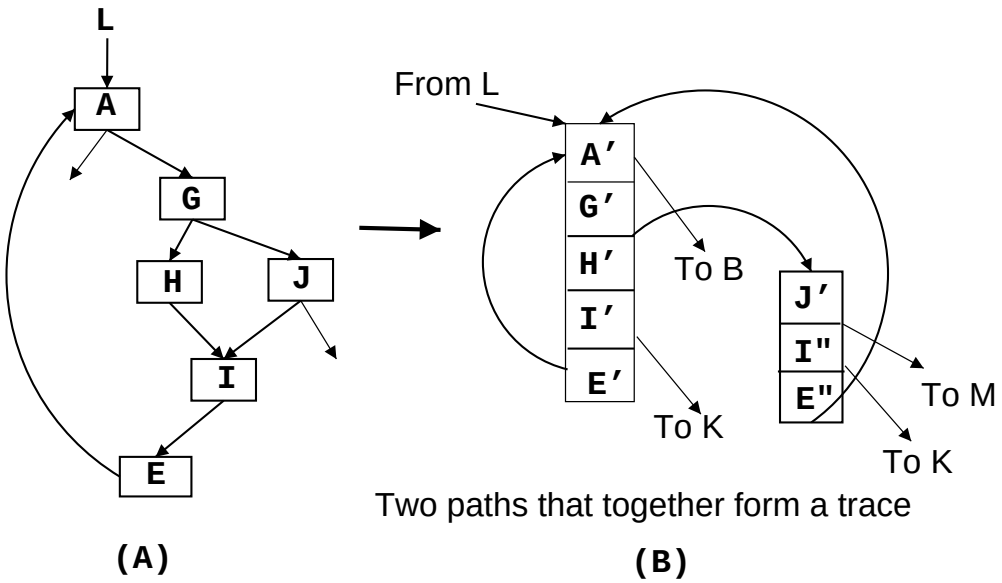

Figure 5.13: Deploying traces in software trace cache. In B, a tarce has been formed from two paths. There is a single entry to the traces placed in software trace cache. Every exit from a trace goes back to the original code.

5.4 Cross-procedural Tracing

Recall that SLI uses a path register to record the path information in a loop region. The technique of using path register is easily extended to capture paths that cross function boundaries. In a loop that executes a large number of times and makes function calls from within the loop body, it is desirable to create traces out of the loop body that also capture the path traversed through the called functions. Such larger traces provide a better opportunity for code optimization. The traces that cross function boundaries are called as interprocedural traces. In the following we describe how a type of functions, called inlinable functions, can be traced for forming interprocedural traces.

The set of functions that do not have any cycles are called as inlinable functions. Recall that the FLI was placed only on backedges of functions. Since inlinable functions do not have any backedge, there would be no FLI in an inlinable function. An inlinable function can have several paths within the function. However, every such path must be acyclic. Some of these paths might be part of loop that frequently makes a call to the inlinable function.

Figure 5.14 shows the creation of interprocedural path starting at A. In the figure, the function f() is inlinable and its conditional branches have been instrumented. Notice that the function f() has only the path-register instrumentation. The instrumentation of the inlinable function is simple: every conditional branch is instrumented. Notice that there is no llvm_sli_count_path() or llvm_sli_loop_exit() inserted in the instrumentation of the inlinable

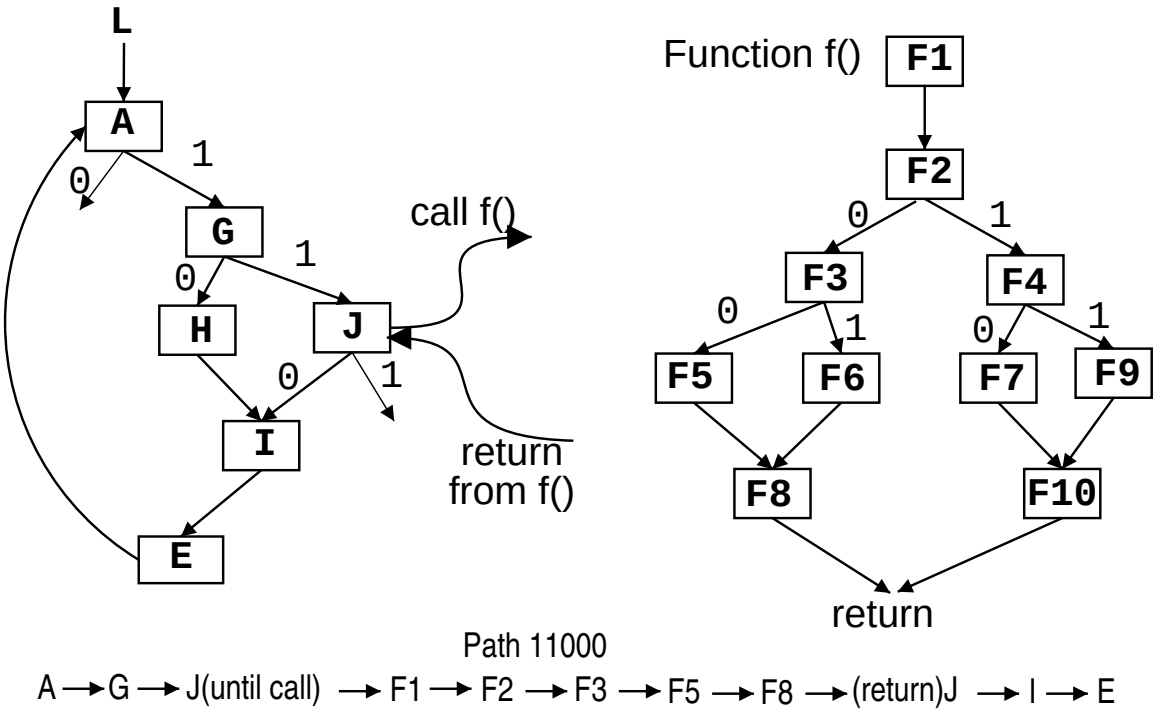

Figure 5.14: An example of interprocedural path being constructed from SLI.

function. This is because any path in an inlinable function is a part of the path in a loop of some SLI.

Figure 5.15 describes how an SLI region must be handled for interprocedural paths. The SLI region gets instrumented when triggered by FLI, and during the SLI instrumentation, all inlinable functions in the SLI region are recursively instrumented. Whenever an inlinable function is found that has not been instrumented already, it is instrumented and placed in the SLI cache. If the called functions are not inlinable, the path register is saved before the function call, and restored after the function call. This ensures that the path register has information only for paths that go across inlinable functions. When an inlinable function is instrumented, any inlinable functions inside it can also be instrumented provided they do not form a recursion. If a recursion is detected, the path register is saved and restored.

5.5 Adapting to Phase Behavior

Notice that in order to keep the overhead of the FLI low, we removed the FLI for a loop region when the iterations of a loop region exceeded a threshold. However, the FLI is *the only* way to trigger SLI and then subsequent formation of hot traces. So on one hand, FLI and SLI are costly, and so they must not execute too frequently. On the other hand, presence

```
CREATE_INTERPROCEDURAL_INSTRUMENTATION(SLIRegion SLI)
 1  HashTable seenFunction
 2  for all called functions x in SLI
 3  do if seenFunction[x] ≠ true AND x is inlinable
 4        then
 5              set inlinedFunction
 6              inlinedFunction ← {}
 7              INSTRUMENT_FUNCTION(x, inlinedFunctions)
 8              seenFunction[x] = true
 9        else  save path register before call instruction
10               restore path register after call instruction
```

```
INSTRUMENT_FUNCTION(Function f, set s)
 1  s ← s ∪ {f}
 2  instrument and link function f
 3  HashTable seenFunction
 4  for all called functions x in f
 5  do if seenFunction[x] ≠ true AND x is inlinable AND x ∉ s
 6        then INSTRUMENT_FUNCTION(x, s)
 7        else  save path register before call instruction
 8               restore path register after call instruction
```

Figure 5.15: Algorithm to recursively instrument inlinable functions for a loop region in SLI

of FLI is the only way to detect hot regions.

Also note that if FLI persists in the original code forever, the instrumentation strategy would be able to capture phase behavior quite naturally. Whenever a loop region executes very frequently, its corresponding traces would be generated. When that happens, the book keeping associated with the loop region, such as the frequency of its iterations, can be initialized to zero. Now if the hot paths for the loop region stay hot, the execution of those paths will occur through the trace cache placed traces for the region. On the other hand, if a different set of paths are now hot as compared to the original traces in the loop region, the executions will exit the trace cache traces, and execute out of the original code. Execution from the original code will trigger llvm_first_trigger(), which was the FLI for the loop region, and this would in turn lead to formation of traces for newer hot paths.

However, leaving the original FLI permanently can be expensive when the loops have erratic behavior. This is because no clear hot paths might exist in the loop, and so no traces might be formed for a loop region. Also, the FLI is quite expensive, so even if it executes a small percentage of total loop iterations, it could still cause a major slowdown of the overall program execution. To avoid the expense of FLI, and to also capture the phase behavior in programs, we ensure that the FLI for any loop region executes a fixed number of times and is then removed. It is again placed back periodically, so as to capture any changes in program behavior. This period for any specific loop region is increased if the loop region does not have phase behavior. For instance, when a FLI for a loop region is removed, it is placed back after 1 interval of time. Now if there is no phase change in the loop, then this FLI is again removed, and this time placed back after 2 intervals of time, and so on.

We make use of a timer interrupt to periodically look at a set of addresses, and place back the FLI at some of those addresses. Whenever a SLI causes formation of traces or rejection of SLI when no hot traces are detected, the address for FLI is placed in a priority queue. The priority queue is sorted in increasing order of a global time associated with each entry of the priority queue. For every FLI location, a counter t is maintained for the number of intervals after which the FLI should be placed back. When placing the location of FLI in the priority queue, we place the current global time, plus the interval t. Thus on every timer interrupt, the interrupt routine increments the current global time, and pulls out of priority queue all address locations for FLI that have time less than or equal to the current global time. Use of priority queue makes this lookup faster and allows eficient implementation of interrupt routine.

Chapter 6

The Software Trace Cache and the SLI Cache

Software trace cache and the SLI cache keep the optimized traces and the SLI code respectively. Both follow same addition and removal schemes. However, both operate on separate memory regions. The memory regions are disjoint so that the traces placed in trace cache have better spacial locality. In the following, we describe the organization of trace cache. The SLI cache has similar organization.

6.1 The Organization of Software Trace Cache

Codes placed in the trace cache are linked with the original code by putting a branch in the original code to the trace cache code. This implies that no two traces in the trace cache can begin with the same address in the original code. A trace can contain multiple paths as described in section 5.3. Every exit from any trace goes back to the corresponding location in the original code. In the figure 5.13(B), the exit from A' in the trace cache goes to B in original code. Traces inside the trace cache are not linked to each other. This makes both addition and removal of traces from the trace cache quite efficient. When removing a trace, the branch from the original code to the trace cache is removed, and the original code in its place is placed back.

The trace cache manages a fixed sized preallocated memory region. Since the trace cache is of fixed size, a code replacement strategy must be used. The code replacement scheme must be both *efficient* and *effective*. Efficient implies the trace cache management must have low overhead at run time. Effective implies that as far as possible, only cold traces get replaced.

Trace cache management differs from the normal OS-level memory management. The

traces are of variable size, and so cannot be allocated fixed size fragments. Also, the traces must be allocated a contiguous segment of memory for efficient code execution. Non-contiguous allocation would hamper fetch from instruction cache, for example. The goal in choosing a memory allocation and replacement policy for traces, therefore, is to reduce fragmentation, and to capture *spacial locality* of execution: the trace most recently executed is potentially still hot.

In our implementation, the trace cache is assumed as a circular buffer. The traces are allocated in order, and removed in order. That is, the first trace is allocated in the beginning of the circular buffer, the next trace after that, and so on. When a new trace can not be added to the trace cache because of limited space, the oldest allocated trace is removed. If still more space is needed, the next oldest trace is removed from the trace cache. This is done until a contiguous memory region can be allocated for the new trace.

In [HS02], several trace cache management schemes are discussed. It is shown that the simple scheme used in our implementation performs as well, and often better than other schemes.

6.2 Trace Cache Memory Management

```
ALLOCATE_MEMORY(Size N)
 1  if ∃ node [a,b] in freelist s.t. b − a ≥ N − 1
 2     then if b − a = N − 1
 3             then erase node [a,b] from free list
 4             else  change [a,b] in freelist to [a + N, b]
 5     else  No node can be allocated

FREE_MEMORY(Address Addr, Size N)
 1  find first node [a,b] in free list s.t. Addr < N
 2  if No such node is found
 3     then insert [Addr, Addr + N] at the end of freelist
 4     else  if a = toAddr + N
 5             then change [a,b] to [Addr, b]
 6             else  insert new node [Addr, Addr + N − 1] before[a,b]
```

Figure 6.1: Algorithm to manage a free list of memory regions

```
Map  existingTrace
Queue  listOfTraces
Map  originalCode

ADD_TRACE(StartAddress addr, Trace t)
 1  if existingTraces has trace with addr
 2     then remove existingTraces[addr]
 3  while Memory manager does not have space for trace
 4  do REMOVE_TRACE(Dequeue from listofTraces)
 5  Allocate memory for traces
 6  Enqueue trace to listofTraces
 7  existingTraces[addr] ← trace
 8  originalCode[addr] ← instruction at address addr
 9  place branch at addr with target as beginning of trace

REMOVE_TRACE(StartAddress addr)
 1  trace ← existingTraces
 2  free memory for trace
 3  Remove trace from listofTraces
 4  erase trace from existingTraces
 5  instruction ← originalCode[addr]
 6  replace instruction at address addr
 7  erase instruction from originalCode
```

Figure 6.2: Algorithm for adding and removing traces

The trace cache uses a single, contiguous region of memory, and allocates it to traces on demand. When it runs out of memory, it removes the oldest trace that exists in the trace cache. The memory manager keeps a list of free memory regions. Every element of list is of form [a, b]. The list is kept sorted, such that if [a,b],[c,d] are two adjacent nodes in the list in that order then $b < c$. The algorithm for allocating and freeing memory regions using the freelist is shown in figure 6.1. Figure 6.2 shows the algorithm for addition and removal of traces from the software trace cache. The SLI cache uses execatly same principles.

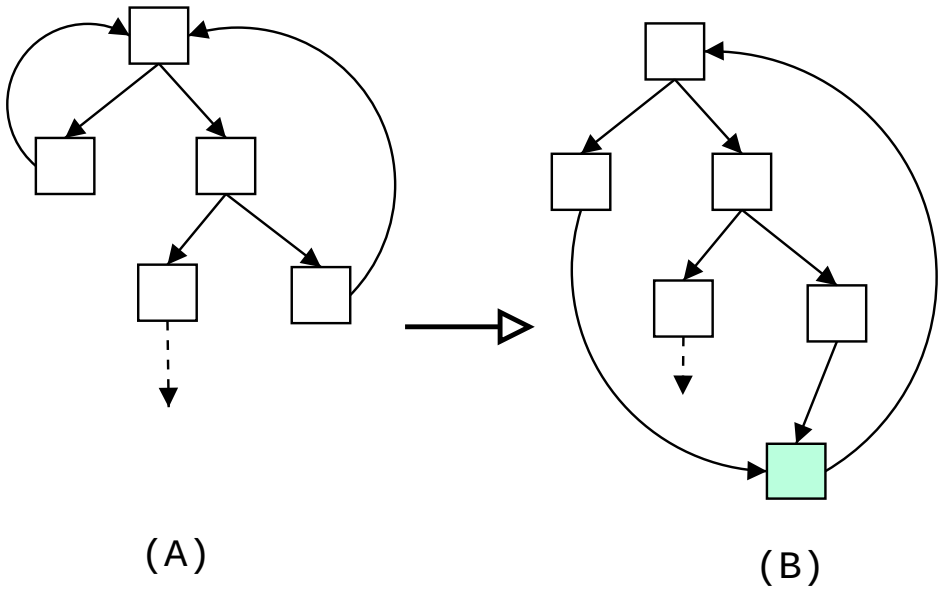

Figure 6.3: An example showing code transformation to reduce trace cache conflicts.

6.3 Reducing Trace Cache Conflicts

Recall that no two traces in the trace cache can have the same start address in the original code. Whenever a trace is added to the trace cache that has the same start address as another trace in the trace cache, there is a conflict. In this section, we describe a simple transformation to the program code which reduces trace cache and SLI cache conflicts. This transformation is performed just before the FLI instrumentation in the static code.

Figure 6.3A shows a loop region with two loops starting at the same node. Both the loops would have their own FLI on their respective back edges. If both the loops are hot, they would conflict in both SLI cache and the subsequent trace cache. Moreover, if the two loops execute alternately, one of the loops would act as an exit for the SLI region of the other loop. This would result in no traces being formed for either of the loops. This problem can be solved by the simple transformation shown in figure 6.3B. In this transformation, a common node is created that jumps to the target of the conflicting backward branches. The conflicting backward branches are converted into forward branches to the new node. Notice that now only one FLI would be placed for the two loops in the example. Also, the two loops would now together form a much bigger SLI region that is likely to yield better traces.

Chapter 7

Results

In this section we evaluate the effect of our instrumentation strategy on overall program performance. A good instrumentation technique for online FDO must have the following features:

- It must have low overhead. This implies that the program performance should not suffer because of instrumentation.
- It should be able to capture traces that form a major fraction of overall program execution.
- It should have low overhead for programs where it can not discover hot traces.

We look at the performance and overhead of instrumented programs in section 7.2. We show that by selecting traces and executing them out of trace cache, we gain performance even when the traces are not optimized. In section 7.3, we look at the fraction of time spent by a instrumented program in executing out of trace cache. A large percentage of execution from trace cache shows the effectiveness of instrumentation in capturing hot traces. In section 7.4, we look at the effect of varying input size on the performance of instrumented programs. The longer the program runs, the lower is the overhead and there is higher gain in performance. In section 7.5, we discuss the sensitivity of instrumentation on the choice of thresholds for SLI and FLI.

7.1 Experimental Setup

For evaluation of results, we compare a instrumented program binary with the uninstrumented binary produced by the LLVM system compiler for SparcV9. The instrumented program is further linked with runtime libraries that handle runtime generation of traces.

As discussed in earlier sections, the runtime system uses several preset parameters. For the results in this section, the chosen parameters are shown in figure 7.1.

Parameter	Value
FLI_THRESHOLD	30
SLI_THRESHOLD	50
SLI_EXIT_THRESHOLD	15
MAX_PATHS_IN_TRACE	6
MIN_ITERATION_FRACTION	90
Size of SLI cache	120KB
Size of software trace cache	120KB

Figure 7.1: The choice of parameters for testing the instrumentation framework.

7.2 Performance and Overhead of Instrumentation

Benchmark(time in secs)	% overhead	% coverage	% speedup
179.art(587)	5.5	87.13	9.1
Stanford-queens(270)	5.3	91.5	8.8
Olden-em3d(135)	9	94.1	-0.8
sieve(48)	3.87	94.06	7.7
heapsort(105)	9.3	93.1	2.9
llubenchmark(96)	1.1	96.4	12.3
Ptrdist-ft(76)	8.1	95.1	-3.5
Olden-tsp(15.4)	4.61	77	3.4
Ptrdist-ks(61)	10.1	53	-7.1
Olden-bisort(160)	3.1	62.3	-1.4
Fhourstones(27)	7.1	51	-3.1
183.equake(1431)	6	21	-0.99
Olden-power(81)	0.6	3.1	-0.2

Figure 7.2: Summary of results: overhead is the total percentage overhead incurred due to instrumentation. Coverage is the percentage of total execution time spent in executing out of trace cache. Speedup refers to the percentage boost in execution of the benchmark when its traces are executed out of the trace cache without optimization, as compared to the execution of uninstrumented program. Time shown with a benchmark is total duration for which a benchmark was executed.

Figure 7.2 shows the summary of results. In the figure, overhead is the total overhead incurred by a program due to instrumentation. This is measured by executing the program, forming SLI regions, creating traces, placing the traces in the trace cache, but executing the original code instead of the code in the trace cache. The speedup column shows the percentage boost or slow down in the performance of the instrumented code as compared to

Benchmark	sli codes	traces	avg #instructions/trace	avg #of paths/trace
179.art	51	39	33.72	1.13
Stanford-Queens	3	2	78	3.50
Olden-em3d	10	6	28.83	1
sieve	3	3	15.33	1
heapsort	3	2	95.50	2.59
llubenchmark	7	4	22.25	1
Ptrdist-ft	7	4	9.25	1
Olden-tsp	6	5	36.40	1.20
Ptrdist-ks	21	11	26.91	1.36
Olden-bisort	2	1	30	1
Fhourstones	10	3	135.53	1.67
183.equake	16	11	31.73	1.09
Olden-power	9	7	138.50	1

Figure 7.3: Instrumentation statistics: The second column shows number of loop regions that are instrumented with SLI. Third column shows the number of traces that get generated. Fourth column is the average number of instructions per trace. Last column is the average number of paths per trace.

the performance of uninstrumented code. A positive number indicates a percentage boost in performance, and a negative number shows a percentage slowdown as compared to the uninstrumented program.

Even though traces are not optimized, the performance in several cases is better than the performance of uninstrumented programs. Again, for benchmarks that did not yield good traces, the performance penalty is quite low. We observe that on average, the programs have 2% speedup in overall performance. This speedup comes because of two reasons. First, the captured traces constitute a high fraction of program execution. Second, because a trace has simpler control flow than a normal program code, it has better locality and thus improves instruction fetch in processors. In Figure 7.4, we look at execution of instrumented program when execution is always from the original code. It shows the overall overhead of instrumentation, with no benefits accruing from executing out of trace cache. We notice that even though overall overhead is large in some cases, the execution out of trace cache is able to ammortize that cost. We also notice that for certain benchmarks, such as power, the overhead is significantly smaller. However, for power, the program slowdown is also small. This implies that even though the tracing framework could not do a good job, it did not suffer large penalties either.

Figure 7.3 shows statistics of trace generation for the chosen benchmarks. It shows the number of SLI codes that were generated as a result of FLI, and the number of traces that

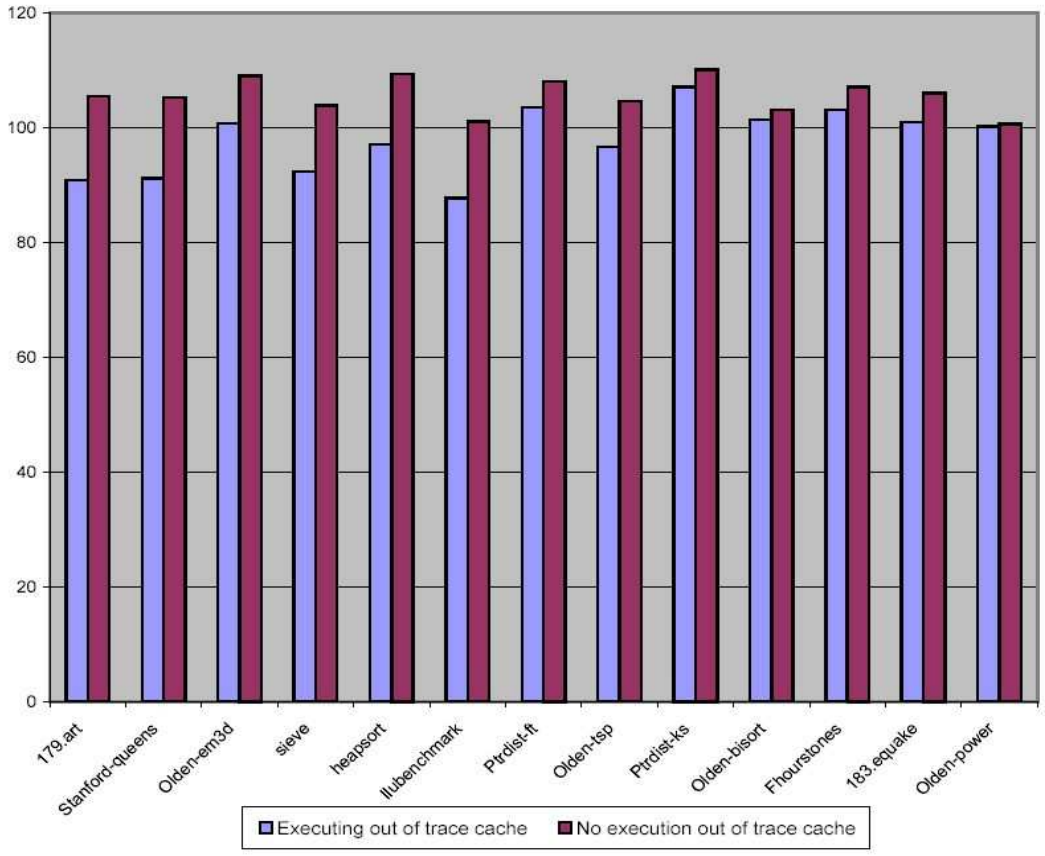

Figure 7.4: Figure showing the execution time of instrumented programs normalized against the time for uninstrumented programs. For every benchmark, it shows two bars. The first bar shows execution time when the generated traces are executed out of trace cache. For the second bar, the program generates the traces, but does not execute those traces from inside the trace cache.

are eventually formed from execution of those SLI codes. It also lists the number of paths on average that together formed a trace.

7.3 Coverage of Traces

In figure 7.2, the coverage column shows the percentage of total time spent executing the code out of the trace cache. A higher coverage indicates the success of instrumentation in capturing paths in hot loops. A low coverage implies that the instrumentation could not locate any clear hot paths in loops. A low coverage could be because of two factors. First, the program may be call intensive and may not have many loops in the program. Second, in the loops that exist, there may be too many paths that are generated frequently, and so there may be no paths that stand out for trace formation. In figure 7.5, coverage is shown

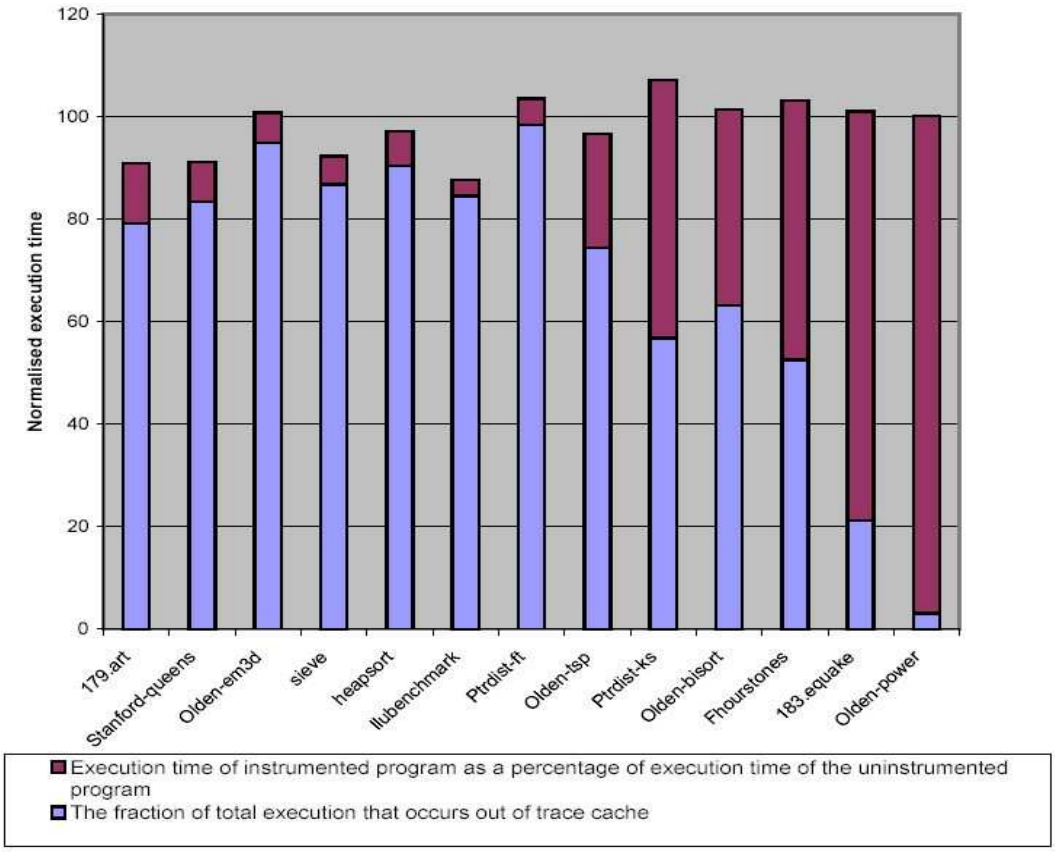

Figure 7.5: Figure showing the execution time of instrumented programs normalized against the time for uninstrumented programs. It also shows the percentage of total execution that occurs in traces placed in the trace cache.

as a fraction of overall program execution. For programs that get a boost in performance because of trace generation, we notice that there is a high coverage. Also, the programs with very low coverage seem to have negligible effect of instrumentation in their overall execution time.

7.4 Effect of Input Size on Program Performance

In figure 7.6 and figure 7.7, we show how the input size affects overall program performance. Observe that as duration of a program execution increases, the program seems to perform better. This is because for a small input size, the program suffers overhead of trace formation, but doesn't sufficiently exploit the execution of traces that get formed. On larger inputs, there is higher execution from trace cache, and the cost of instrumentation as a fraction of overall execution also becomes lower. The figures validate this effect: with longer execution, a larger fraction of program execution is out of trace cache, as seen by higher coverage in

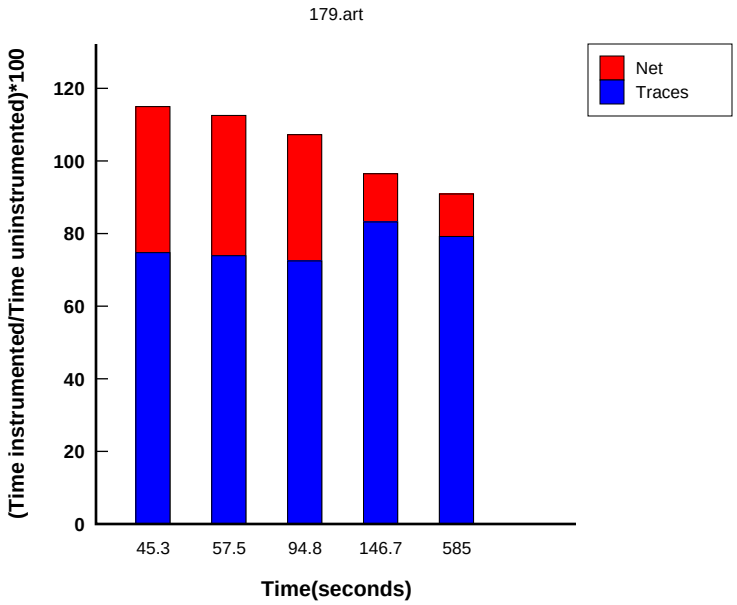

Figure 7.6: Effect of varying input size on instrumented program for benchmark 179.art. *Net* indicates the net execution time of the instrumented program as a percentage of the execution time of uninstrumented program. *Traces* shows the fraction of time spent by instrumented program in executing out of trace cache.

the bars.

From figure 7.3 we see that 179.art produces 51 loop regions with SLI code, and out of those loop regions, 39 lead to formation of traces. Thus, it spends considerable amount of time in formation of traces that are not sufficiently exploited for small input sizes. With larger inputs, however, there is a significant speedup of upto 10% as benefits of trace execution begin to accrue.

7.5 Effect of FLI and SLI Thresholds on Execution

Figure 7.8 and figure 7.9 show the effect of choice of FLI and SLI thresholds on program performance. In the diagram, the benchmark 179.art is executed with a small input size. When FLI threshold is small, the loops with FLI would be converted into SLI sooner, and so there would be lesser penalty for FLI function calls. However, with small FLI, even non-hot loops would be instrumented with SLI, and SLI creation is expensive. Also, the SLI code is heavily instrumented and is expensive to execute. Longer execution of SLI helps to locate the correct hot paths. However, a shorter execution of SLI can also lead to correct hot paths, in which case the cost of SLI is minimal. The diagrams show that there is better

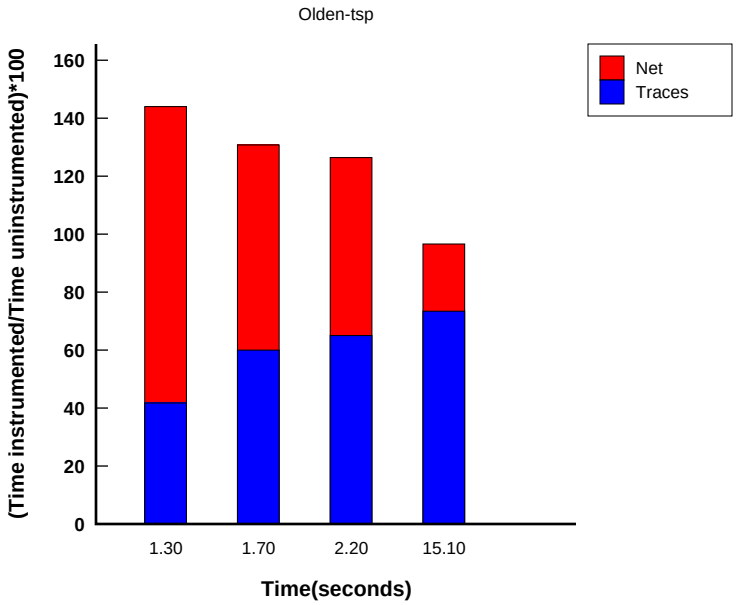

Figure 7.7: Effect of varying input size on instrumented program for benchmark Olden-tsp. *Net* indicates the net execution time of the instrumented program as a percentage of the execution time of uninstrumented program. *Traces* shows the fraction of time spent by instrumented program in executing out of trace cache.

performance as FLI threshold is increased and SLI threshold is reduced for the benchmark 179.art. When both FLI and SLI thresholds are large, there is a significant slowdown. Also, when FLI threshold is very low and SLI threshold is very large, the performance is worst. This is as expected: a large number of non-hot loops would be instrumented with SLI, and the SLI codes would execute for longer time, resulting in a large performance penalty.

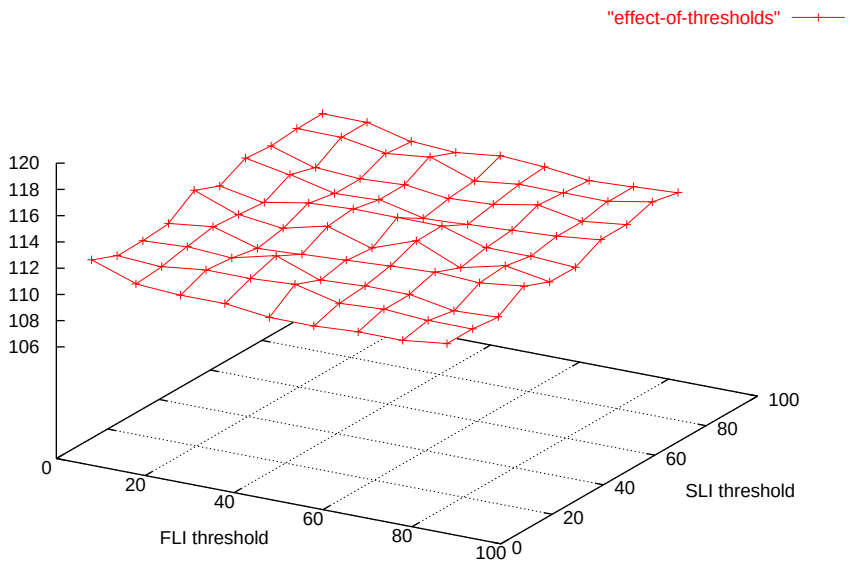

Figure 7.8: Effect of FLI and SLI threshold on overall program performance of 179.art. The z axis shows the total time taken by the instrumented program as a percentage of time taken by the uninstrumented program. Lower is the percentage, the better is the performance.

FLI threshold	SLI threshold	% slowdown
1	1	13.60
1	3000	61.40
3000	1	10.80
3000	3000	46.60

Figure 7.9: Effect on performance of 179.art with very small and very large threshold values for FLI and SLI.

Chapter 8

Implementation Details

In this chapter we describe some of the implementation issues of the interprocedural tracing framework. We first briefly describe the major components in the system. The runtime system makes several assumptions about the code. These assumptions are described in section 8.2

8.1 Major Components of Tracing Framework

The instrumentation framework makes use of the following major components:

Memory Manager The memory manager manages a fixed area of memory. It is initialized with a pointer to a memory region and size of the region. It allows two operations: getMemory and freeMemory. The algorithm for these operations is described in figure 6.2.

Virtual Memory Manager The virtual memory manager provides an interface to the sparc/solaris proc file system. In solaris, the virtual memory used by the program can be read and written using the proc file system. The virtual memory manager component hides the details of reading and writing into program virtual memory.

Trace cache The trace cache allows runtime addition of traces for an executing program. It uses the memory manager to obtain a memory region where the traces are deployed. The trace cache addresses every trace by the starting address of the trace in the original program code. Thus, it can contain only one trace at a time with a given start address. When traces are added to a system, the trace cache writes a branch at the start address of trace in the original code. It stores the instruction that was overwritten by the branch. When it runs out of memory, it removes the oldest trace in the system in

order to create more memory. It keeps on deleting traces in this order until a sizeable chunk of memory is available to allocate a new trace. The traces are allocated in a circular buffer. While removing a trace, the branch from the original code to the trace is removed, and the code that was removed to place the branch is reinstated. Thus, trace removal is transparent to the user of tracecache. The trace cache library is linked to the instrumented program during its compilation to native code.

Function table emitter This component is a pass on LLVM bytecode. For every internal function in a program, this pass creates a obtains a pointer to the function and inserts the pointer in a table. The table is then inserted as a global value in the LLVM bytecode. At run time, this table gets filled with the starting address of every internal function. This address information is used by the Mapinfo component to provide various mappings from the LLVM system to the runtime system.

Inlinable table emitter This component is also a pass on the LLVM byte code. For every internal function in a program, this pass checks to see if there are any backedges in a function. If there are no backedges in a function, the function is said to be inlinable. This information is then generated in a table, and placed as a global value in the program bytecode.

Mapinfo The LLVM code generator stores a one-one map from the LLVM basic block to machine basic block. At run time, this map is read once and cached for any future use. Since the map is one-one, it provides information in both directions: given a LLVM basic block, it provides the start address of the basic block, and vice versa. It also provides one-one mapping information for LLVM function, and its starting machine address in a program. This component is linked to the instrumented program binary for use at runtime.

FLI instrumentation FLI instrumentation is a pass on the LLVM bytecode. It detects the backward branches in a CFG, and instruments the backward edge with a function call to llvm_first_trigger. This pass also narrows down the set of back edges that get eventually instrumented. For every basic block containing a backward branch, every other basic block that it dominates are left uninstrumented.

SLI instrumentation SLI instrumentation occurs at runtime. It is invoked by the FLI present in a program. This component creates the SLI for a loop region, and adds it to the trace cache.

Trace generator The trace generator uses path register, and the frequency of path executions to create a trace. The trace can consist of multiple paths: the trace generator tries to generate a single trace out of the traces such that there is minimal duplication of code among the set pf paths that get generated. It does so by looking at the most common prefixes among traces.

8.2 Runtime Invocation of Instrumentation

The llvm_first_trigger() function call, which is part of the code generated by a static compiler, invokes formation of SLI and subsequent traces at runtime. Since the FLI does not take any arguments or return any values, the following techniques are used to progressively generate SLI, and then traces.

1. SLI system uses LLVM CFG for analysing and instrumenting a loop region. For this purpose, the static system must store the LLVM bytecode with the program during code generation. The LLVM bytecode is read at runtime to create the LLVM functions and modules.

2. Many a times during SLI and trace generation, register values need to be saved and restored. In order to facilitate this, we create two slots on top of the function stack that are never used by the static code generator. The slot creation is done by the static code generator.

3. The llvm_first_trigger reads the return value from the function. In sparcV9, this is the register i7. The following set of assumptions are made regarding the nature of runtime code:

   (a) It is assumed that the static compiler would have placed the unconditional branch after this function call

   (b) The branch and the call must be in the same LLVM basic block.

   (c) Every llvm basic block has a one-one mapping to a machine instruction basic block at runtime.

   (d) Instructions have not been moved across basic blocks after the basic block maps were generated.

   We now try to read instructions beginning with the instruction at the return address, and keep reading ahead instructions till a branch is found. Note that with the assumptions mentioned above, a branch will always be found. Also note that there is no

way to assertain this run time, and so the above assumptions are important for this implementation.

4. The target of the branch is the *root* of the SLI region. Also, the target address *must be start of a basic block*. We now use the Mapinfo component to get the LLVM basic block that corresponds to the target address. Also, we find the LLVM basic block which would have contained the branch. This forms the BB. Using the root and the BB, the instrumentation can be generated for the SLI region as described in figure 5.6.

5. When a trace is generated as a result of execution of SLI code, the SLI code in the cache is not deleted. Instead, a branch is placed from the top of the SLI code to the top of the trace. Also, addition of trace cache automatically adds a branch from the original code to the trace. The branch placement form SLI code to the trace is also essential because otherwise, the program may continue executing out of the SLI code for much longer time, and this would slow down the overall performance.

6. Before generating SLI code for any code region, it is checked if there is no SLI code for that region already present in the SLI cache. SLI code would get deleted from the SLI buffer only as a result of code replacement policy of the SLI cache (when the cache is full), or when there is a conflict with some new SLI code which is being added to the SLI cache. Therefore, there is always a chance that the SLI code that was generated earlier, might still exist in the SLI cache. This saves lots of computation since SLI generation is expensive. The same ideas also hold true for trace additions to the trace cache.

7. Notice that there are two separate cache regions: one for SLI code, and other for traces. Both have exactly same functionality. However, they are kept separate for an important reason. A SLI code will *always conflict* with any trace that gets generated out of that SLI code, since both the SLI code and the trace would have same start address in the original code. This implies that addition of trace would remove the SLI code. However, this removal will break code execution, since trace is generated as a result of function call from within the SLI code. Therefore after the trace generation, execution must return back to SLI code. Keeping the two caches separate takes care of this problem.

8.3 Sparc Dependent Implementation Features

We make use of the following SparcV9 based features in our implementation, which would need to be adapted onto the architecture where the framework is implemented.

1. In our implementation, we conveniently use function calls for instrumentation. In particular, three functions are used for instrumentation: llvm_first_trigger, the function llvm_sli_count_path, and llvm_sli_loop_exit. However, sparcV9 places several restrictions on code generation which would limit arbitrary placement of functions among a sequence of instructions at run time.

   (a) The code generator does not save and restore register values for llvm_first_trigger function. This functions internally preserves the values on entry, and restores the values on exit. Also, in sparcV9, a call instruction modifies the register o7. Thus save and restore of this register, if it is being used across a call to this function, is done by the code generator.

   (b) The function llvm_sli_count_path is inserted *to replace* the llvm_first_trigger. Note that this function is placed at the end of a simple path, and so path register g1 need not be saved and restored inside this function. Also, since the register o7, if used, would have been already saved and restored by the code generator around the FLI, no save and restores for o7 need be created. This function must save and restore all values internally, just like the llvm_first_trigger.

   (c) The llvm_sli_loop_exit function gets inserted where there was no function earlier in the program. Therefore, it must take care that it does not destroy any register values at the program point where it is inserted. Before inserting this function, we first save register o7, and restore the register o7 just after the function call. Inside the function, we save and restore all other register values.

2. Register g1 is used as path register for sparcV9 implementation. This register is left unused by the code generator. In sparcV9, g1 is considered as volatile across function calls. So even though any function produced by the LLVM code generator would not use it, an external function might, and so destroy the value in this register. Just before every function call in the SLI code, it is checked if the function is inlinable. If it is not, then this register is saved before the function call, and restored after the function call. Note that if a function is inlinable, it *must be* an internal function that was compiled by using LLVM code generator.

Chapter 9

Conclusion

In this work we have shown an effective way to detect hot traces for feedback directed runtime optimization. We introduced a path-register based instrumentation strategy that allows easy detection of interprocedural paths. We also showed how a potentially costly instrumentation can be used as long as the instrumentation itself does not execute frequently. Using a single call instruction for the first level of instrumentation makes it easy to switch off instrumentation, and later put it back on if needed. The instrumentation strategy discussed in this paper allows to discover traces without interpretation in any stage of program execution. In our implementation of this work, we benefitted by using a compiler system that makes it possible to delegate part of the runtime optimization work onto the static compiler.

Splitting overall instrumentation into two levels allows several significant benefits. The expensive part of the instrumentation (SLI) is invoked only when a loop region seems to have a high chance of yielding hot traces. Also, the initial instrumentation has a small foot print, and therefore can be compiled into the program binary. The static compiler can be used to narrow down the regions of program for FLI, thus further reducing the instrumentation overhead.

The path register based instrumentation, used in SLI code to detect hot paths in a loop, is an efficient and simple way to locate paths. In RISC or EPIC architectures, the path register can be a machine register. Thus it can be implemented with a cost of two instructions on every conditional branch. The path register based instrumentation also gets easily extended for paths that cross function boundaries: and we have shown how it can be used to capture loops that traverse multiple inlinable functions.

The SLI instrumentation distinguishes between paths that iterate through the selected backedge (the simple paths), and the ones that do not. For SLI a loop region is selected, and only portion of the program within the loop is instrumented. To make sure that the instrumentation for the loop yields correct traces, we keep a count of exits from the SLI

region. This allows us to exclude generation of traces from outer loops when an inner nested loop executes more frequently. The SLI instrumentation also tells us if more than one path is hot, and we combine all the hot paths in a loop together to form a trace.

We have also proposed a simpler trace cache organization that is partly possible because of how we form a trace: if there are more than one hot paths that start at the same point, we generate a single trace out of them and add them to the trace cache as a single entity. Since our traces are not connected to each other, they allow very simple cache management. Notice that the hot paths in a loop, which are most likely to be intertwined in their executions, would already be connected to each other in the trace cache.

Our results have shown that the two level instrumentation strategy can capture a high percentage of program execution without incurring large overheads. Also, for programs that this strategy does not do well, such as for call intensive codes, the overhead is kept very low.

The implementation of this work benefits from close interaction with the static compiler. Apart from the intitial instrumentation, the static compiler also stores bytecode that allows efficiently creating CFG at runtime. However, use of bytecode at runtime is not unique to the LLVM system. Systems such as Java JIT compilers and CLR also make use of bytecode at runtime in similar ways.

References

[ABD+97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone, 1997.

[Adl98] A-R. Adl-Tabatabai, et al. Fast and effective code generation in a just-in-time java compiler. In *Proc. 1998 Conf. Prog. Lang. Design and Implementation",*, May 1998.

[AFG+00] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Conference on Object-Oriented*, pages 47–65, 2000.

[AHR02] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of java. In *17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, November 2002.

[Bal96] Vasanth Bala. Low overhead path profiling. Tech. report, HP Laboratories, 1996.

[BDB99] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization. Tech. Report Report #HPL-1999-77, HP Laboratories, 1999.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conf. on Prog. Lang. Design and Implementation*, pages 1–12, June 2000.

[BL94] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[CL99] R. Cohn and P. Lowney. Feedback directed optimization in compaq's compilation tools for alpha, 1999.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[FPP98] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[GBF97] Rajiv Gupta, David A. Berson, and Jesse Zhixi Fang. Path profile guided partial dead code elimation using predication. In *IEEE PACT*, pages 102–, 1997.

[GBF98] Rajiv Gupta, David A. Berson, and Jesse Zhixi Fang. Path profile guided partial redundancy elimination using speculation. In *International Conference on Computer Languages*, pages 230–239, 1998.

[HS02] Kim Hazelwood and Michael D. Smith. Code cache management schemes for dynamic optimizers. In *Proc. Workshop on Interaction between Compilers and Computer Architecture*, Boston, MA, Feb 2002.

[Kla00] A. Klaiber. The Technology Behind Crusoe Processors, 2000.

[KS73] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.

[LA02] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[Lar99] James R. Larus. Whole program paths. In *In Proceedings of the SIGPLAN 99 Conference on Programming Languages Design and Implementation*, 1999.

[Mic] Microsoft Corporation. The .NET Common Language Runtime. See web site at: http://msdn.microsoft.com/net.

[MTB+01] Matthew C. Merten, Andrew R. Trick, Ronald D. Barnes, Erik M. Nystrom, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50:567–589, 2001.

[PL01] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, Jun 2001.

[RJSS97] E. Rotenberg, Q. A. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proc. 30th Int'l Symp. on Microarchitecture*, pages 138–148, Dec 1997.

[Smi00] Michael D. Smith. Overcoming the challenges to feedback- directed optimization. In *Proc. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, Boston, MA, Jan 2000.

[YS98] Cliff Young and Michael D. Smith. Better global scheduling using path profiles. In *Proc. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 115–123, November 1998.