ABSTRACT

Title of dissertation: FOO'S TO BLAME: TECHNIQUES FOR MAPPING PERFORMANCE DATA TO PROGRAM VARIABLES

Nickolas Jon Rutar, Doctor of Philosophy, 2011

Dissertation directed by: Professor Jeffrey K. Hollingsworth,
Department of Computer Science

Traditional methods of performance analysis offer a code centric view, presenting performance data in terms of blocks of contiguous code (statement, basic block, loop, function, etc.). Existing data centric techniques allow various program properties to be mapped directly to variables. Our approach extends these data centric mappings. Just as code centric techniques allow lower level objects like source lines be mapped up to functions, our inclusive technique allows low level data centric operations like computations on scalars to be mapped up to complex data structures like those found in scientific frameworks. Our system utilizes static analysis to collect information about the program that can be combined with runtime information to perform data centric program analysis. By pushing most of the analysis to pre-run and post-mortem, we can minimize the amount of data collected at runtime. This allows us to perform less instrumentation and also minimizes program perturbation. It also allows us to collect information that would not be possible with existing techniques.

We present two applications of this analysis. The first application of our analysis is targeted at mapping performance data to high level data structures with multiple levels of abstraction. We create extended data centric mappings, which we call variable blame, that relates data centric information to these variables.

The second application is a method for mapping cache miss information to variables. Existing approaches for this analysis rely on explicit hardware support and extensive program instrumentation. By utilizing our analysis and applying software heuristics, we are able to lessen those requirements.

We apply both of these analyses to applications and show what performance information can be provided by our analysis that can not currently be determined. We also discuss how we can use that information to improve program performance.

FOO'S TO BLAME: TECHNIQUES FOR MAPPING PERFORMANCE DATA TO PROGRAM VARIABLES

by

Nickolas Jon Rutar

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair/Advisor
Professor M. Coleman Miller, Dean's Representative
Associate Professor Jeffrey S. Foster
Professor Adam A. Porter
Professor Alan L. Sussman

Dedication

To my wife - Colleen,
For support, love, support, and lunch money.

To my parents - Merry Lou & Louis Rutar,
For Encyclopedia Brittanica and the Apple 2c.

To Cashew and Pinot,
For never asking when I was going to graduate.

Acknowledgments

I would like to begin by thanking my advisor, Dr. Jeffrey K. Hollingsworth. His support and patience were greatly appreciated and his guidance was invaluable to my research. This document would not have been possible without him.

I want to thank my advisory committee. Each member of the committee played a special part in shaping my graduate career, both academically and personally.

I have been very lucky to have had a great support system in school without even having to look outside of my research group – Ray Chen, I-Hsin Chung, Tugrul Ince, Mike Lam, Jeff Odom, Rahul Patibandla, Geoff Stoker, Mustafa Tikir, Ananta Tiwari, & Chadd Williams. Thank you to Ray for always trying to get us to hang out as a group outside of A.V. Williams. Thank you Tugrul, for our long conversations about who is the biggest 'slacker' between the two of us while carpooling. Thank you Geoff, for the spirited political debates on our many walks to Taco Bell. Thank you Mike, for dongles with a smile. Thank you Tikir, for quotes about goats. Thank you Chadd, a man whose skill for dishing advice is only matched by his skill for serving chili. Finally, thank you A.T. for making me take breaks that forced me to go out in the sun every couple hours, saving me from madness and rickets. Also, thank you to all the other friends I made outside of my group, mainly from playing IM sports. I take complete blame (it's a common theme in this document) for any and all losses to the business school.

Thank you to my family. Mom, Dad, Kris, Gina, Chad; thank you for your support but be warned. Tech support from a Ph.D. is not cheap and no, I don't take checks. Aunt Deb, thank you for being the teacher for my first C.S. class.

Finally, Colleen, I can not thank you enough for all your understanding and support. I promise no thinking about work 24/7! My mind on my honey and my honey on my mind.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program performance data has traditionally been presented to the user in terms of code regions. The most common way to aggregate the data is in terms of time spent in a function although statement, basic block, and loop nest data are also common. The measured metric can be wall time, total cycles, cache misses, floating point operations, etc. This code centric view is helpful in identifying hot spots in a program. However, it is not the only way to analyze and present performance data. The presentation of data to users in terms of program variables is another useful approach that has less frequently been utilized. While not as commonly used as code centric methods, data centric analysis augments code centric data with additional insights into program behavior.

In performance profiling, analysis can be done in an inclusive or exclusive manner. Exclusive analysis concerns only one level of abstraction. Inclusive analysis provides drill-down information. The primary motivation for this work comes from the fact there are no existing performance analysis techniques that utilize an inclusive, data centric analysis. Table 1.1 shows examples of how data would be represented for inclusive and exclusive analysis for both data centric and code centric methods. The classic profiling technique is the flat, exclusive, code centric view. This shows how much time is spent in the actual function and disregards how much

	Code Centric	Data Centric
Exclusive	foo() 20% bar() 30% baz() 50%	fooVariable 30% barField 30% bazSubField 20%
Inclusive	foo() 100% ↪bar() 80% ↪baz() 50%	fooVariable 80% ↪barField 50% ↪bazSubField 20%

Table 1.1: Inclusive and Exclusive Analyses for Code and Data Centric

time is spent in functions that are called from the profiled function. The inclusive, code centric view requires more information to be gathered at runtime, but gives richer data that shows time spent for each calling context.

Prior work [14] has been done for mapping data centric events such as cache misses to variables in an exclusive manner. This type of analysis attributes performance information to the allocated memory region assigned to a variable, but does not attribute data over all the fields and sub-structures in a complex data type. Previously, inclusive analysis for complex data types did not exist. This type of analysis and its applications is the core of this dissertation and leads us to our thesis.

The thesis of this dissertation is that inclusive data centric analysis can be utilized for better program understanding and to improve program performance. Furthermore, the underlying approach used to provide the inclusive data can also be used to expand existing data centric analysis techniques across new architectures. To validate this thesis, we introduce two techniques which are capable of mapping program data to variables in an inclusive manner. These techniques are "variable blame" and "approximate data centric" analysis. We present results in

this dissertation from using these techniques to profile multiple programs.

Programs that contain multilevel abstractions are the main target applications for our inclusive approach. Programmers think in terms of data objects (linear systems, PDEs, matrices), not functions and low level arrays. These objects often are inherently distributed and contain calls to message passing libraries that are completely hidden from the user, masking both data motion and the parallelism. Unfortunately, when these abstractions are introduced it becomes more and more difficult to diagnose performance and correctness issues using conventional means. The higher level the abstractions, the harder it is to figure out the lower level constructs that map to them and subsequently discover where performance problems are occurring. This affects both application programmers as well as the designers of the libraries when they try to understand application performance characteristics and tune the performance of their software. We believe a profiling environment can represent performance data in terms of these abstractions, mainly the instantiations of these abstractions in the form of program variables. The unique feature of our tool and techniques is the ability to automatically combine and map performance and debugging data from complex internal data structures (such as sparse matrices and non-uniform grids) to higher level concepts.

Our primary approach for our inclusive data centric analysis is the variable blame approach. Variable blame records data flow information from the program and uses it to map performance information to variables. Blame determines what explicit and implicit data flow was utilized to determine a variable's value. During event-driven sampling, if a sample occurs that falls under the set of data flow op-

erations that may have ultimately contributed to a given variable's value, then the variable is "blamed" for that sample. Variable blame is defined in Chapter 3 with a calculus that formally represents how blame is calculated. As blame is a profiling technique, we created a prototype tool that uses blame analysis as its core means to profile programs.

We use the blame based tool to perform blame analysis experiments on various benchmarks and applications. We examine how blame analysis specifically compares against existing code centric approaches in the uniqueness of the data it presents. We present how information given by blame analysis can assist in improving program performance. We also present how blame analysis scales across parallel systems. The experimental results are presented in Chapter 4.

Our inclusive, approximate data centric analysis expands upon existing exclusive, data centric approaches that involve direct measurements [13]. Existing approaches use hardware counters to assign cache and TLB misses to variables. However, these approaches rely on specific hardware support and extensive source and/or binary instrumentation. We show in Chapter 5 how we are able to use software techniques to approximate these measurements. These approximations maintain rank order for the profiled variables while removing the hardware limitations and instrumentation constraints.

The main contributions of this dissertation are:

Variable Blame Calculus

We present a language independent, formal definition of how to calculate variable blame.

Blame Tool and Experimental Results

We present the tool and use it to create case studies showing how the values determined from variable blame can help with program understanding and compare those values to those found by code centric means. We also use variable blame values to improve program performance over different hardware configurations.

Approximate Data Centric Analysis

We introduce an approach to compute data centric statistics without the need of the traditional hardware support utilized by the direct measurement methods.

Chapter 2

Related Work

An important concept in this work is the mapping between different abstraction levels at the source level down through to the compiled code. To have any kind of working tool for performance analysis, there is an explicit need to associate the chosen metric to a source level construct so that improvements to the code can be made. This mapping becomes more complicated with parallel programming languages and languages with complex abstractions and runtime optimizations. The related work section begins with discussion of prior work on mappings. We examine both data centric mappings and those mappings that utilize internal abstractions to represent both code and data centric program elements. Since inevitably these mappings are plugged into some kind of program analysis software (as is the ultimate destination of the mappings for this work) a small survey of related performance analysis programs is discussed. There is a rich history of tools that do performance analysis. Rather than provide an exhaustive comparison, we have chosen a mixture of tools that are most closely related and that also represent a breadth of features previously explored. Finally, there are a series of enabling technologies that are important to our implementation.

2.1 Mapping

The two kinds of mappings that we are interested in are "data centric" and "abstraction based." The data centric mappings directly map performance data to variables. However, they are exclusive and are limited in the types of performance metrics that can be assigned to the variables. The abstraction based mappings are more complex and can have code centric and data centric elements mapped to their abstractions at any point in the program. These mappings are not strictly data centric and involve binding code centric regions to variables at different parts of the program. Our mappings are an amalgam of these two approaches. Our mapping is hierarchical like the abstraction based approaches. However, it is strictly data centric and bound to variables at every point in the mapping.

2.1.1 Data Centric Mappings

Existing data centric mappings are exclusive, and try to map data centric events like cache and TLB misses to the variables whose access caused the miss [9, 13,40,45]. The common limitation for all of these approaches is the special hardware features needed to run the different kinds of analysis. The first hardware requirement is that an effective address can be retrieved from the processor at the time the measurement is taken. The effective address can sometimes be reconstructed by decoding the instruction that caused the miss to determine the addressing mode and binding the values in the applicable registers to determine the address.

The second hardware limitation concerns the skid factor. The skid factor

is an artifact of instruction sampling, where a triggered event (such as a cache miss) is attributed to an incorrect instruction. One cause of skid can be due to out-of-order execution, however, the skid factor is still an issue for processors with in-order execution [18]. For in-order execution, skid is a function of the size of the pipeline and any of the instructions in the pipeline at the time of the event may be the cause. For time based metrics, skid is not as significant a factor. This is because the instruction the event was attributed to was valid in the instruction pool. However, for data centric events the skid factor can mean assigning the event to an instruction that is accessing a different memory region or even assigning the event to an instruction with no memory accesses. Certain architectures have hardware in place to negate the skid factor and will return the precise IP (instruction pointer) for the event and effective address of the miss when running in that mode [20,33,34]. In cases where an architecture does not have this hardware in place, a user is completely unable to use the existing approaches. As HPC systems move to less complex CPUs (i.e. stream processors & GPUs), the type of hardware support required for data centric measurement may become less available [48].

A very common approach for attributing cache misses to variables is to generate samples using memory based hardware counters and assign counts to the data structures responsible for those misses by using the effective addresses. This method was introduced by Buck [14]. His approach triggers samples based on cache misses and uses the effective address of the data being accessed at the sample point to increment a counter for the responsible variable or dynamically allocated block of memory. This approach has the requirement that the architecture have the proper

skid negating hardware available. Besides the hardware limitations, this approach also has the minor disadvantage of introducing some program perturbation. In order to map the effective addresses, you need to keep a record of all of the allocations and frees in the program for dynamically allocated memory. For allocation routines, this involves instrumentation and performing a stack walk to gather context sensitive information at runtime.

Itanium is one architecture with hardware support [33] for this effective address matching technique and is the system Buck's approach was implemented on with his tool Cache Scope [13]. On Itanium, the proper hardware counters are available and the effective data address of a specific cache miss can easily be retrieved.

HPCToolkit [46] has recently added data centric profiling to their tool [40]. Their approach is very similar to the approach used by Cache Scope. Like Cache Scope, they record the allocations and frees in the program. They also rely on hardware support to negate skid and to provide a precise effective address of the memory that triggered the event. Their tool uses the PEBS (Precise Event Based Sampling) feature [34] on select Intel chips and the IBS (Instruction Based Sampling) feature [20] on certain AMD chips.

Our "approximate data centric" approach closely resembles that of Cache Scope and the data centric capability of HPCToolkit. However, our approach is usable on architectures without specific hardware that negates skid. Our work also eliminates the need for monitoring all allocations and frees within the program.

Other tools that use data centric mappings include StatCache and Memphis. StatCache [9] is a probabilistic model of the cache. It does this by wrapping all

loads and stores in the program, which can create a large overhead. The information gathered at runtime is used to simulate the memory hierarchy and to apply the post-mortem model to predict cache performance. This information is used to improve data locality. Memphis [45] is a data centric toolset that is limited to the AMD architecture due to its reliance on IBS. It focuses primarily on finding NUMA-related problems.

2.1.2 Abstraction Based Mappings

Irvin introduces concepts involving mapping between levels of abstraction in parallel programs with his NV model [36] as utilized by the ParaMap [37] tool. In the NV model, a noun is any program element that a performance measurement can be attributed to. This can include program components such as functions, source lines, or loops. It can also include arrays and variables within the program. A verb consists of any action taken by or performed on a noun. This would include things like the actual execution of a source line or function in the program, or an assignment operation between two variables. Sentences have exactly one verb, all participating nouns, and the associated cost of the action. The set of sentences at any given software or hardware layer forms a level of abstraction. The mappings then run between sentences at different levels of abstraction. The mappings can be determined either through static analysis prior to execution (static) or at runtime (dynamic).

Static information consists of all the information gathered before the program

executes. An example of this type of mapping would be source lines (single or multiple) mapped down to lower level routines that execute those line(s). Because this all is computed before runtime, it can be stored in whatever form the user chooses such as in the actual image, a database, or some external file.

Dynamic information contains the exact same mapping information as static, but the information is derived at runtime. The SAS (Set of Active Sentences) is the data structure that allows dynamic mapping of concurrent sentences between layers of abstraction. The SAS pertains only to those sentences that are active at that point in the program. As sentences become active/inactive at their respective abstraction layers, they are added/removed from the SAS. Furthermore, any concurrent sentences in the SAS are dynamically mapped. The SAS is used to answer "performance questions" based on one or more sentences within the SAS.

The SAS approach has three self-described limitations. The first is that the approach does not handle asynchronous activation of sentences. This could mean that certain sentences at a higher level could potentially not be available to be mapped to the proper lower sentences so performance data would not be properly bubbled up.

The second limitation is that sentences that are dynamically activated yet ignored by the SAS create larger execution costs than is necessary. This limitation is based on the fact that performance questions are raised after the set of active sentences exists, so notifications from unused sentences within the SAS would be ignored but the cost to raise the notification would remain. They acknowledge this could be remedied by adding additional functionality to dynamically remove

unneeded notifications.

The third limitation is that sentences are not ordered in performance questions. The order that the sentences are parsed in a performance question can yield very different performance questions.

The Semantic Entries, Attributes, and Associations(SEAA) [57], a followup to SAS and the NV model, addresses some of the limitations discussed above and adds other features that were not present in SAS. SEAA creates support for user-level abstractions by allowing the definition of semantic entities as both annotations and independent sentences. A semantic entity is an element that is defined solely on semantics, meaning an element can be created and used without it having a direct program construct attached to it (though regular program constructs can still be entities). This is essentially an upgraded model of the sentence abstraction within the NV model, with the ability to add new constructs that was not possible in the NV model. A semantic attribute is a set of semantic information that can be attributed to an entity for qualifying mapping relationships. A semantic association creates a link to cost mapping based on the attributes of the entities.

The SEAA mappings follow many of the same conventions as introduced in the NV model with the additions discussed above. These additions allow a much larger domain of programs that can be mapped. As a result, the model directly addresses some of the limitations that were raised in the NV model. The addition of semantic attributes address the sentence ordering problem. The attributes could be used to attach additional semantic information to the sentences that could allow an ordering to take place.

The problem of asynchronous sentences is also addressed with SEAA mappings. At a given layer of abstraction, sentences may be partitioned into appropriate entities and mapped to their respective associations using the appropriate attributes. In the cases of asynchronous execution, the associations will map back up to the appropriate entities and will report their respective PMEs (performance measurement entity), regardless of the order of execution. The resolution of the asynchronous sentence problem creates an environment that offers more accurate bookkeeping than the NV model.

The SEAA model, however, does introduce more overhead than the NV model to achieve the more accurate modeling. Whereas the SAS aspect of the NV model looks at all of the currently active sentence and answers performance questions based on that information, the SEAA model examines the attributes for the individual entities which creates more bookkeeping.

2.2 Profiling Tools

Tools that try to represent the information in terms of aggregates and statistical representations of program metrics are referred to as profiling tools. Profiling tools are able to represent entire runs of the program in a variety of ways depending on the granularity of the measurements and the metrics that are being measured. A trivial profiling tool takes as input a program and returns the wall time that program takes to run. In terms of scope, tools may use the whole program or may drill down to specific functions, basic blocks or individual source lines. In

terms of the metrics being measured, time is the most commonly measured element, but other program attributes such as cache misses, floating point operations, bytes transferred during communication, and other hardware or software attributes may be used. Many hardware counters can be measured by utilizing APIs such as PAPI [11] which create a machine independent abstraction to access the counters on most current microprocessors.

Performance information traditionally has been gathered through instrumentation through the insertion of calipers, either to the source or binary (statically or dynamically). A caliper is a pair of instrumentation points that allow the profiler to gather performance data (often time) by taking measurements at both points and calculating the difference. The alternative method of instrumentation is sampling based. Sampling interrupts the program at certain intervals and samples the state of the program at these interrupt points. In the case of calipers, the area of interest is delimited by instrumentation calls that measure metrics before and after the measured region. While the caliper based approach gives exact measurements, there are certain overhead and side effects from program instrumentation. By using the sampled information, tools can create approximations for the same performance metrics utilized by the traditional caliper-based profiling.

2.2.1 Caliper Based Instrumentation

The main differences between profiling tools in this category is the mechanism for instrumentation and the presentation of the data collected. In terms of

instrumentation, the application can either be instrumented at the source or binary level. Source instrumentation allows easy insertion at specific source lines but also has some disadvantages. The program has to be recompiled after instrumentation is added and the instrumentation process may interfere with compiler optimizations within the program. Binary instrumentation takes place after the image has been generated (no recompilation necessary) so compiler optimizations would remain intact. However, this does not mean that the program will necessarily perform the same as if the instrumentation had never occurred. The instrumentation calls themselves may interfere with some of the performance traits the tool itself is trying to record [43]. Binary instrumentation may occur statically or at runtime. In both cases, the main deterrent for binary instrumentation is the extra level of complexity required for the tools in handling instrumentation of the images. Extensive knowledge of the platform specific binary formats and the compilers that generated the code are required for successive binary instrumentation.

The choice of when to do the instrumentation is just one aspect of the instrumentation process. The decision of what program elements to instrument and measure is also important. Some tools choose to represent a control flow graph of the program by instrumenting at the basic block level. Others instrument at the function level. Most use some combination of different program constructs chosen by the user depending on the desired granularity. There are also several tricks used by various programs to combat the overhead involved in instrumentation. The Tuning and Analysis Utilities (TAU) tool [58] avoids the instrumentation of low level functions called multiple times, limiting the overhead of measurement to a manage-

able level (less than 1%). The PT project instruments selectively along nodes in the control flow graph based on a heuristic created from data from prior runs [7].

One of the most popular active profiling tools is the aforementioned TAU tool from the University of Oregon [58]. Its popularity comes partly by the fact it is available on most platforms and supports a variety of languages, including Fortran, C, C++, Java, and Python. TAU also handles language extensions such as OpenMP and MPI implementations on the supported platforms. The framework for TAU is divided into three layers: instrumentation, measurement, and analysis. Instrumentation is primarily source based, but dynamic instrumentation is also supported by using Dyninst [12]. The measurement phase is primarily profiling based but also offers support for tracing. The analysis phase has a variety of options for viewing and examining the data.

SvPablo [55] is a language independent profiling tool. One of the features that separates it from most profiling tools is a GUI that allows a user to interactively instrument the source program. One of the primary goals of the project was the ability for the user to learn one program that would allow similar experiences over different languages. This is accomplished by storing all metrics as a hierarchy of Self-Describing Data Format (SDDF) records. Three groups of record descriptors are in SDDF: mapping, configuration, and statistic. Each statistic based on performance analysis is mapped to one of the constructs that was instrumented.

2.2.2 Sampling

Sampling based profiling gathers information by periodically recording the program state and uses that information to estimate the overall performance of the given metric that was sampled. Because instrumentation is limited to enabling sampling within the program, most sampling tools require no recompilation of the executable or access to the source. The caveat is that many sampling tools require that the application had been compiled with debugging symbols for source line resolution or compiled with a profiling flag enabled in the case of prof [26] and gprof [25]. Sampling is lighter weight than full caliper based instrumentation with the accuracy tradeoff that the calculation of information is based on a small number of data points versus the direct measurement of caliper based approaches. Most sampling techniques either sample the program counter or the current call stack. The program counter permits the reverse mapping to a region of code. The call stack will give information about the exact context of the sequence of code, but has more overhead involved since more information needs to be recorded and stored.

The frequency of sampling can be set by a number of means. One way is to sample based on a set time interval. This was the primary approach for older sampling tools and is the approach of prof [26]. Another approach is to sample based on a set number of instructions issued. This is the approach for sampling tools such as VTune and DCPI [3]. Another approach, called event sampling, involves sampling based on metrics measured by hardware counters such as cache misses. This approach usually involves having dedicated hardware support to count these

events and issue interrupts when the counter overflows a pre-determined limit. Most current architectures have this support. HPCToolkit [46] and Speedshop [56] allow this kind of sampling.

One of the oldest, lightweight sampling tools is prof [26]. It is available as a standard application on many Unix/Linux platforms and requires a compilation flag to enable a program to be profiled. Prof simply interrupts at a given time interval and uses that information to approximate the relative distribution of time over the entire program. However, for prof the amount of time assigned to each procedure does not include the time used by procedures further down the calling tree. The tool gprof [25] is an extended version of prof that also counts the number of times that each arc in the program's call graph is traversed. Thus, the time reported for gprof does report the time for procedures down the calling tree.

HPCToolkit [46] is a set of tools that uses sampling for its measurement phase, but whose primary contribution is a series of visualization tools to present the data to the end user. At the core of the toolkit is the 'hpcview' tool which correlates profiling data to a hierarchical program context (file, procedure, loop, line). On Linux, the 'hpcrun' tool does the actual sampling and uses the PAPI library to access hardware counters. The event interrupts utilize the hardware counter overflow feature discussed above. On non Linux platforms, vendor supplied tools are utilized. A third tool, 'hpcprof,' maps the profiling data collected by hpcrun back to source lines.

Intel has a set of performance profilers that use sampling. Their first tool to use sampling was Intel VTune [32]. VTune originally had simple sampling techniques

based on instructions issued, but now additionally supports event based sampling. The Intel PTU (Performance Tuning Utility) [31] was more recently built as a compliment to the VTune tool. The Intel PTU utilizes the Precise Event Based Sampling (PEBS) system. The PEBS system allows profiling by both the Instruction Pointer(IP) and the data address. With event based sampling, the PEBS record can give the exact instruction that caused the interrupt.

The HP Digital Continuous Profiling Infrastructure (DCPI) [3] is a tool that has limited support in its current incarnation, but uses sampling in ways that differ enough from other sampling tools that bear mentioning. Originally developed at Digital, it takes continuous samples of entire systems, including the operating system, user programs, the kernel, drivers, and shared libraries. A database is updated after every program is run. Positive features of DCPI are that no recompilation is required, profiles can be provided all the way down to the instruction level, and there is minimal overhead.

2.2.3 Hybrid

There are some tools that offer choices to the user on whether to use the traditional instrumentation method or a more lightweight sampling method. SpeedShop [56], developed by SGI, is one of those tools. In terms of sampling, SpeedShop allows event based sampling and sampling based on instructions issued. Depending on the type of sampling involved, SpeedShop either uses the PC or call stack based sampling to resolve performance to program locations.

SpeedShop also supports caliper based instrumentation for certain situations. For example, it supports an "ideal time" experiment that instruments the program using pixie [56]. This analysis calclulates the cost per basic block and the number of basic blocks within the executable. This information is then used to calculate the "ideal" time for the program. Ideal is rarely attainable, however, as this form of profiling does not take into account issues such as cache misses, stalls, etc.

2.3 Other Tools

2.3.1 Tracing Tools

Tracing tools are similar to profiling tools in that they are post-mortem performance analysis tools. However, while profiling tools aggregate information to summarize the performance for a given scope, a tracing tool will record all events of interest that characterize the program to some log. By looking at the full log at the end of a program, a user can recreate the execution of the program.

While the amount of information generated by tracing can obviously be of great help to a user, there are some obvious issues that are not present in a profiling technique. First, the storage overhead for these logs can be tremendous. This is especially true for distributed and long running programs. Second, even when the information can be properly stored there has to be an intuitive way in place to examine the large amounts of data. Many tracing tools use specialized visualization tools designed specifically for parsing and visualizing tracing data. Some of the more popular of these visualization tools include Paraver [50], Vampir [63], and

Jumpshot [68].

The Kit for Objective Judgement and Knowledge-based detection of performance bottlenecks (KOJAK) [24] uses program traces to try to discover bottlenecks within parallel programs. KOJAK uses its own CUBE viewer to visualize the data. Like its geometric namesake, CUBE has three dimensions in its presentation of data: the metric, source code location, and node/thread location. KOJAK is also somewhat unique in the fact it primarily relies on manual instrumentation by the user. It does provide some mechanisms for automatic instrumentation in certain contexts, but manual instrumentation is still expected to take care of the majority of the trace points.

2.3.2 Online Analysis Tools

Online analysis tools are different from post-mortem tools in the fact they take measurements of the program while it is running and allow the user to manipulate the program during execution to modify performance. This can be accomplished by statically instrumenting the program with hooks that allow runtime modification of program elements to influence program behavior. Dynamic instrumentation can also be used to modify these programs on the fly.

Paradyn [47] is an online tuning tool that aids in discovering program bottlenecks. Paradyn searches for bottlenecks with its W^3 search model [29]. The W^3 search model asks *why*, *where*, and *when* the application is performing poorly. The "why" axis is represented in the form of potential bottlenecks with hypotheses and

tests and is searched iteratively by going through each hypothesis and testing if it is true. The "where" axis is a collection of logically independent resource hierarchies. The search over the "where" axis is also iterative and involves traversing each of the resource hierarchies. The "when" axis has the user testing hypotheses within different time intervals throughout the program's execution. Paradyn uses Dyninst to perform dynamic instrumentation for its online tuning. This allows Paradyn to insert and delete snippets of code on the fly depending on the current area of investigation for the W^3 model.

2.4 Supporting Analyses

We present related work in areas of analysis that are similar to some of the components used by our approach.

Dataflow analysis is utilized in many areas. Guo et al. [28] dynamically records data flow through instrumentation at the instruction level to determine abstract types. This is similar to the explicit data flow relationships we use in our blame analysis. Other work dealing with information flow in control flow statements is similar to the implicit data flow relationships we use [44, 64].

Pointer and alias analysis is a large research field in and of itself. Pointer analysis traditionally relies exclusively on static analysis, whereas our work most closely matches other non-traditional work that combine static and dynamic information for the generation of points-to sets [4, 27, 49]. Transfer functions are a way in alias analyses to propagate information about side effects and potential mappings

up from callee to caller. [35, 67]. However, the utilization of transfer functions for our blame mapping differs significantly from the use in alias analysis. The primary difference is that transfer functions in alias analysis serve as a means to propagate pointer information, whereas we utilize transfer functions for blame mappings. Another difference is that in alias analysis, the use of transfer functions is performed strictly based on information gathered by static analysis. This may lead to situations where a transfer function is forced to operate based on incomplete information. Our transfer functions utilize both static analysis and runtime information. This allows us to have more complete information about the context for each call site. We discuss transfer functions in further detail in Section 3.1.4.

A related area of work to pointer analysis and transfer function generation is escape analysis [19,51]. Escape analysis examines a function to determine whether a variable/pointer can "escape" and be accessed outside of the scope of that function or region in the code. We utilize escape analysis to determine which variables within a function can be used to represent the blame for our own transfer functions.

2.5 Techniques involving 'Blame' Terminology

There is other work that uses the term "blame." Although this work is not related to our work, because of the similarity in terms we would like to explicitly address that there is no connection between our work and other "blame" work.

One area of work that uses the term "blame" is Findler's work on contract checking [22]. In his work, a series of contracts are in place for methods within a

program. Blame is assigned when a run-time check determines a contract has been violated. In this context, the source of blame is the particular contract (usually attributed to a method) that has been violated and the entity assigned the blame is the individual responsible for providing the element of the program that violated the contract. In our system, the source of blame is performance metrics and the blame is assigned to program variables.

Chapter 3

Variable Blame

Blame is an inclusive data centric approach that determines what explicit and implicit data flow was utilized to determine a variable's value. Explicit operations are represented by the chains of data writes within the program. Implicit operations are based primarily on control flow. During event-driven sampling, if a sample occurs that falls under the set of data flow operations that may ultimately contribute to a given variable's value, then the variable is "blamed" for that sample.

To illustrate the basic concepts behind blame we examine the small C snippet in Figure 3.1. We argue that the purpose of this entire snippet is to populate the value of *c*. Since our data collection is sampling based, any sample that occurred within this code snippet would be blamed on variable *c*. This small example case would be contained within a single basic block. This chapter will show how blame would be propagated across multiple basic blocks in a single function and how blame would be mapped to variables across function calls using whole program analysis.

```
1   int a, b, c;
2   a = 5;
3   b = 6;
4   c = a + b;
```

Figure 3.1: Small Code Snippet

Code centric approaches attribute values to basic blocks or functions based on the sampling location. The difference in our approach is that we combine the sample's location and our analysis to attribute the sample to one or more variables instead of attributing the sample to code regions. As is the case with traditional code centric sampling, samples can be triggered by timers or can be event driven through hardware counters. Blame can be attributed to variables using any metric that can be measured on a system and has an appropriate hardware counter associated with it. This includes cycles, instructions issued, and cache misses.

We will present the formal definition of blame in terms for a single variable for one run of a program with sampling enabled. Let S be the set of all samples (represented as the sampled statement) gathered for the run of the program. For a given sampled statement s within S, let W be the set of all statements containing a write to the memory region allocated by variable v, the aliases of v, and all fields of v. For the fields, this includes all sub-fields within the hierarchy of v. This includes both pointer and non-pointer fields. The blame set for v is the union of all the statements in the backward slices [66] for each of the statements in set W,

$$BlameSet(v, W) = \bigcup_{w \in W(v)} BackwardsSlice(w)$$

Variable v is blamed for a sample in the cases where s is a member of the $BlameSet(v)$ which we represent with this function,

$$isBlamed(v, s) \ \ \{ \text{if}(s \in BlameSet(v)) \ \ \text{then} \ \ 1 \ \ \text{else} \ \ 0\}$$

The blame percentage for a variable for the entire program is the percentage of samples that can be blamed to a particular variable divided by the total number of samples. This is represented with the following formula,

$$BlamePercentage(v, S) = \frac{\sum_{s \in S} isBlamed(v, s)}{|S|}$$

After we have performed this calculation, we can say variable v is "blamed" for that fraction of whatever metric we used to generate the samples. For example, if we were sampling off of cycles and the blame percentage was x, we would say that v was responsible for the x fraction of all cycles over the course of the program.

The discussion above is only a means to provide a formal definition for variable blame. It is not indicative of how blame is calculated in our system. In the next section, we will formalize how variable blame is calculated with a calculus. We will also present a graphical implementation of the blame calculus. We then present our blame tool that utilizes the graphical implementation for its pre-run analysis, has runtime data gathering, and performs post-mortem analysis on the data we gathered statically and at runtime. We follow this up with a section detailing how those components would be used to perform full program analysis on a small sample program. Finally, we detail how the data would be presented to a user using our tool.

3.1 Blame Calculus

We define a language where all operations are performed on Single Static Assignment (SSA) registers. Loads and stores allow data movement between these registers and higher level variables. These variables are not limited by the SSA requirement and can have multiple loads and stores modifying their values. This language is typical of intermediate representations for higher level languages used in program analysis or compilation. In this language, the blame mappings are blind to the manner in which a variable is allocated. Local variables (stack-allocated), global variables, and heap-allocated variables are all accessed through the load/store operations and are internally handled in an identical manner.

We use the following notation to represent the language:

r ::= register

n ::= integer

v ::= $r|n$

V ::= Variable

Variable blame is based on dataflow interactions, so to formalize the propagation of information we use gen-kill sets. Gen and kill sets represent the subset of elements for each set that are inserted and removed, respectively, for each operation [38]. All sets are initialized to the null set at the entry of the function. Each variable and register maintains an individual copy of its corresponding blame set and supporting sets. The gen-kill sets are calculated at the basic block level and

propagated in standard fashion through the entire function based on the control flow graph for the function. The propagation for each set (for each variable and register) follows the *in* and *out* formulas shown below, where 'n' is a basic block identifier.

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

$$out(n) = gen(n) \cup (in(n) - kill(n))$$

The analysis is a forward, may analysis [1, 2, 38].

The instructions in our language and how the sets are manipulated by these instructions are shown in Tables 3.1 and 3.2. For these tables, the '@' sign represents a pointer dereference. For store operations, the first operand in the statement is the data being propagated and the second operand is the location in memory where the value of the first operand is being stored. Because of the SSA property of registers (they are only written to once), the register ID is used to reference the register itself, and also the statement where the register is introduced with a write. For this reason, although the members of the sets are of different types of elements, register IDs will appear across all of the sets. The 'store' instruction is handled differently than other instructions when dealing with the gen-kill sets. It is the only instruction for which a register ID is not the identifier placed in the set since no register is written to, only memory locations. For store instructions, we construct a label for the instruction with the source line number (from the original program) that the store maps up to. That label is then placed in the appropriate set(s) in place of the register that

would normally accompany an instruction in our language.

There are multiple cases for dealing with stores depending on properties of the input operands. In cases where the target is a non-pointer variable, we use the rules associated with "store $r, @V_1$." We make a special case for this store because non-pointer variables are the only case that utilizes the Temp Blame Set, discussed below. The other store instructions in the table assume the store target is associated with a pointer variable.

The generic store case, and the one in which blame is propagated the most, is associated with the "store $r, r1$" instruction. For this instruction, we assume $r1$ is a register holding a pointer to some variable. Because of the complexity of the set operations in this function, we make a call to the 'doStore' function to make the appropriate set insertions and deletions. This function is the main link between the primary blame set and all of the supporting sets. The 'doStore' function is discussed in detail in Section 3.1.1.

The final store case, "store $V, @V1$," concerns how we deal with alias relationships. This is not meant to be an exhaustive representation of our alias analysis, but is meant to serve as a base case corresponding roughly to the C code,

```
int * x, * y;

x = y;
```

Our full alias analysis can not be properly represented by the blame calculus language. Because of this fact, the base case is simply meant to show how our set operations react to an alias being generated. The set operations that occur for alias

relations not shown by our calculus follow the same gen-kill relationships as the base alias case. The generation of the aliases themselves is performed by following approaches outlined in work by Wilson et al [67] and utilizing the alias functionality provided by LLVM [41].

The complete list of gen-kill sets used for the calculation of variable blame is listed below. For the Blame Set and Temp Blame Set, the members of the set are individual statements from the language. For all other sets the members are registers and variables.

Blame Set (B)

This is the primary set that is propagated by the gen-kill process. There is no kill action for this set. For registers, this is because of the SSA property. Since registers are written to once, that initial write creates the blame set. For variables, the temp blame set is utilized for transferring blame from variable to variable (and has kill sets). The blame set is only written to when we want the value to be attributed to the final blame set.

Temp Blame Set (T)

This is a temporary blame set utilized for the higher level variables. The reason for this set is to have a new set each time a store writes to a non-pointer variable.

Aliases (A)

The set of calculated aliases for a variable. Each variable has its own copy of A, which changes based on the program point. When we detect an alias assignment, we add that relationship to the set. When that alias is invalidated by another assignment, we kill the old alias relationship. Therefore, depending on the point at which the analysis is being done, each variable will have a different alias set then it might at another program point.

Up Pointer (U)

We maintain an up pointer for every data access from the point of access to the variable (and register) whose data space was accessed.

Down Pointer (D)

The set contained by each variable that contains all of the registers that have accessed its data space.

Field Parent (P)

An up pointer to the containing variable for each field.

Fields (F)

The set of all fields for a given variable.

Control Flow Dependent (CFD)

The set of registers contained within the instructions that may be executed based on the register value in the conditional branch. The calculation of this set of registers is discussed in Section 3.1.2.

	Blame (B)	Up Pointer (U)		Down Pointers (D)	
Statements	**Gen**	**Gen**	**Kill**	**Gen**	**Kill**
$r = v_1$	$r : \{\{r\} \cup \mathrm{B}(v_1)\}$	$r : \{\mathrm{U}(v_1)\}$		$\mathrm{U}(v_1)$:$\{\{r\}\}$	
$r = v_1$ op v_2	$r : \{\{r\} \cup \mathrm{B}(v_1)$ $\cup\ \mathrm{B}(v_2)\ \}$	$r : \{\mathrm{U}(v_1)\}$		$\mathrm{U}(v_1)$:$\{\{r\}\}$	
store $r, r1$	doStore(r, r_1)				
$r = @V_1$	$r : \{\{r\}\}$	$r : \{\{V_1\}\}$		$V_1 : \{\{r\}\}$	
r =load $@V_1$	$r : \{\{r\} \cup \mathrm{T}(V)\}$				
r =load r_1	$r : \{\{r\}\}$	$r : \{\{r_1\}\}$		$r_1 : \{\{r\}\}$	
r =load 100	$r : \{\{r\}\}$				
$@V =$ malloc v_1	$V : \{\{v_1\}\}$		$\forall d$, $\mathrm{D}(V)$ $d : \{\{V\}\}$		$V :$ $\{\mathrm{D}(V)\}$
store $V, @V_1$ (at line l)	V_1:{{'store(l)'}} V:{{'store(l)'}}		$\forall d$, $\mathrm{D}(V_1)$ $d : \{\{V_1\}\}$		$V_1 :$ $\{\mathrm{D}(V_1)\}$
store $r, @V_1$ (Non-Ptr V_1)	$V_1 : \{\ \mathrm{B}(r) \cup$ 'store(l)' $\}$				

	Field Parent (P)		Fields (F)	
Statements	**Gen**	**Kill**	**Gen**	**Kill**
$V \equiv V_1 \rightarrow f$	$V : \{\{V_1\}\}$		$V_1 : \{\{V\}\}$	
$V \equiv V_1.f$	$V : \{\{V_1\}\}$		$V_1 : \{\{V\}\}$	
$@V =$ malloc v_1		$\forall f$, $\mathrm{F}(V)$ $f : \{\{V\}\}$		$V : \{\mathrm{F}(V)\}$
store $V, @V_1$		$\forall f$, $\mathrm{F}(V_1)$ $f : \{\{V_1\}\}$		$V_1 : \{\mathrm{F}(V_1)\}$

	Temp Blame Set (T)		Aliases (A)	
Statements	**Gen**	**Kill**	**Gen**	**Kill**
$@V =$ malloc v_1				$V : \{\mathrm{A}(V)\}$
store $V, @V_1$			$V : \{\{V_1\}\}$ $V_1 : \{\{V\} \cup \mathrm{A}(V)\}$	$V_1 : \{\mathrm{A}(V_1)\}$
store $r, @V_1$ (Non-Ptr V_1)	$V_1 : \{\mathrm{B}(r)\}$	$V_1 : \{\mathrm{T}(V_1)\}$		

Table 3.1: Intraprocedural Explicit Gen Kill Sets

	Blame (B)
Statements	**Gen**
br $r1$, label <true>, label <false>	$\forall a$ in CFD($r1$) $a : \{\ \mathrm{B}(r1)\}$

Table 3.2: Intraprocedural Implicit Gen Kill Sets

3.1.1 Explicit Calculation Involving Pointers

We handle a store to a register holding a pointer by performing the *doStore* function, shown as one of the set operations in Figure 3.2. This function recursively goes up the pointers for the containers (designated as a Container type, which can be either a register or variable). It also recursively goes up the parent field pointers and also traverses the aliases for each pointer.

```
function upSet(Container r,
  Set S)
{
  if r is null
    return
  S.insert(r)
  upSet(r, S)
  ∀a,A(r)
    upSet(r, S)

  return S
}
```

```
function doStore(Register r,
  Container r1)
{
  if r1 is null
    return
  Set S = {}
  r1.B = r1.B ∪ upSet(r, S)
  doStore(r, P(r1))
  doStore(r, U(r1))
  ∀a,A(r1)
    doStore(r, a)
}
```

Figure 3.2: Pseudocode for doStore (and upSet) function

The reasoning behind this function is that because of the inclusive nature of blame, we want to be thorough and visit the entire pointer chain that led us to this access. We also want to traverse the entire data structure this field was a part of and mark each container variable until we reach the top of the field hierarchy.

The Blame Set for each register found in the *doStore* traversal has the return set from the *upSet* call inserted into it. The *upSet* call is similar to the operations in *doSet*, except there is no recursive traversal of the field up pointers. The *upSet* function operates on the register that is transferring blame over. With this element, we want to transfer over the blame from all the aliases and up pointers. However,

we do not want to transfer over blame from other fields, as the computation for those fields may be completely unrelated to the blame that was assigned to the field that we are transferring blame from.

3.1.2 Implicit Calculation

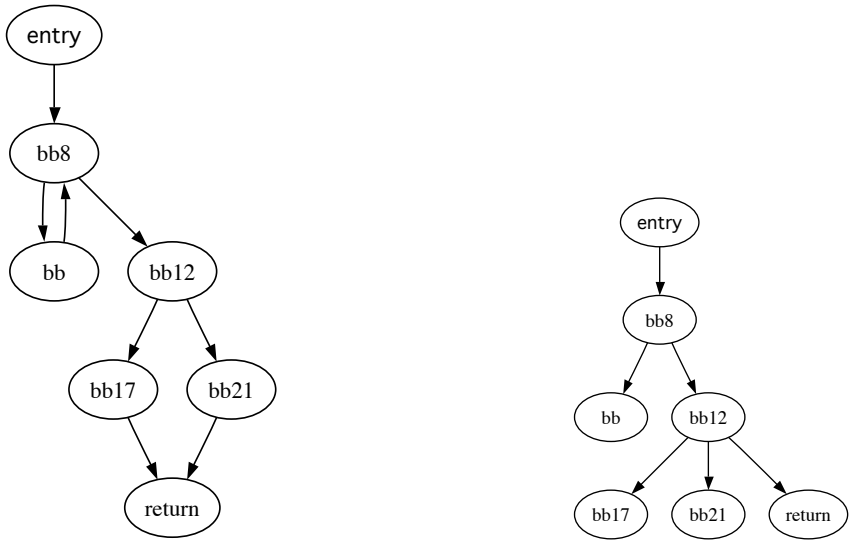
We have primarily discussed how to propagate blame through direct dataflow. Variable blame also takes into account implicit dataflow operations as well. The algorithm for determining the implicit registers utilizes both the CFG and dominator tree for a function.

Given CFG_p as a connected graph for procedure p and is represented by a 3-tuple, (s, B, E_{CFG}) where $s \in B$ is the root node, B is the set of basic blocks, and E_{CFG} is the set of edges.

Given Dom_p as the dominator tree for procedure p and is represented by a 3-tuple, (s', B', E_{Dom}) where $s' \in B'$ is the root node, B' is the set of basic blocks, $s == s'$, $B \equiv B'$, and E_{Dom} is the set of edges.

An important distinction to make before describing our algorithm is even though the set of nodes for the CFG and dominator tree are equivalent, the edges linking them will differ between the two. For example, in Figure 3.3 we see that the dominator tree for vertex "bb12" has three children (bb17,bb21,return) while the CFG for that same node has two children (bb17,bb21).

Figure 3.4 shows three functions for propagating implicit dataflow. One function calculates implicit operations for conditionals, one examines loop operations,



a) b)

Figure 3.3: The a) CFG versus b) dominator tree for function 'bar' in Figure 3.15

and the final function calls the first two with the appropriate parameters for each function.

The conditional function recursively calculates the implicit set for each basic block. For its parameters, v represents a basic block and I is a set of implicit registers. In addition to the instructions, each basic block has a local copy of I, mainly the set of implicit registers that affect it. The function is initially called by passing in the NULL set for I and the root node for the dominator tree as parameter v. The function works by traversing the dominator tree and looking at the number of children that overlap between the dominator tree and the control flow graph. In cases where edges match between the two and the base node ends with a conditional branch, we know that the children are dependent in a conditional operation. We perform additional checks, not shown in the pseduocode, to assure that we are not dealing with control flow edges that are a part of loop operations. Once we establish

```
function genImplicitCond(BasicBlock v,  ImplicitSet I) {
  v.I = I
  if (deg+_Dom(v) == 1)
     genImplicitCond(v_child where v --EDGE--> v_child ∈ E_Dom , I)
  else if (deg+_Dom(v) == 0)
     return
  else if (deg+_Dom(v) > 0)
   {
     t = terminator instruction of v
     t.i = register within t that determines control flow
     for all dominator tree children c of v
        {
          if (v --EDGE--> c ∈ E_Dom and v --EDGE--> c ∈ E_CFG)
             genImplicitCond(c, I ∪ {t.i})
          else
             genImplicitCond(c, I)
        }
   }
}

function genImplicitLoops(Loop L) {
  comp = comparison basic block of L
  t = terminator instruction of comp
  t.i = register within t that determines control flow

  for all basic blocks b of L
  {
    b.I = b.I ∪ {t.i}
  }
}

function genImplicits(Function F) {
  DominatorTree DT = F.dominatorTree()
  genImplicitCond(DT.root(), NULL)

  Loops L = F.loops()
  genImplicitLoops(L)

   for all basic blocks b of F
  {
      for all instructions i of b
         i.B = i.B ∪ {b.I}
  }
}
```

Figure 3.4: Implicit Blame Calculation Pseudocode

the conditional relationship, we transfer the register that determined the control flow for the terminator instruction to the implicit set for the appropriate basic blocks.

The function that handles loops acts by determining which basic block is responsible for the properties of the loop, mainly the one with the compare operation. We grab the terminator instruction from that block and propagate it to all of the basic blocks within the loop in a similar manner to the conditional implicit sets.

The function that calls the loop and conditional functions takes a variable representing the function to be analyzed in as its input. It calculates the dominator tree and all natural loops occurring within the input function. It uses this information to pass the appropriate parameters to the worker functions. Once the implicit sets are calculated, each instruction in the basic block (represented as SSA registers) has the registers from the implicit set transferred to their respective blame sets.

3.1.3 Blame Calculus Example

To illustrate how the sets interact with each other in calculating both explicit and implicit blame, we will now work through a small example program. We will examine a single sample function with no outgoing calls, with the exception of one to *malloc*. The function, both in original C code and represented by our language, is shown in Figure 3.5. A table with the statements and the state of each set after the gen-kill operations are applied is shown in Table 3.3. The blame (for both IR statements and source lines) that is attributed to each local variable at the end of the function is represented in Table 3.4.

```
1   typedef struct {
2     int *i;
3   } StructEx;
4
5   void oneFunc()
6   {
7      StructEx s;
8      int * x, * y, z;
9      x = y;
10     z = 1;
11     if (z) {
12        x[1] = z;
13     }
14     z = 2;
15     if (z) {
16       s.i = (int*) malloc(4);
17       s.i[0] = z;
18     }
19  }
```

a)

```
entry:
1   store y, @x
2   t0 = 1
3   store t0, @z
4   t1 = load @z
5   tB = compare t1, 0
6   br tB, label bb, label bb7

bb:
7   t4 = @x
8   t5 = t4 + 4
9   t6 = load @z
10  store t6, t5

bb7:
11  t7 = 2
12  store t7, @z
13  t8 = load @z
14  tB1 = compare t8, 0
15  br tB1, label bb12, label bb20

bb12:
16  @(se.sX) = malloc(4)
18  t9 = @(se.sX)
19  t10 = load @z
20  store t10, t9
```

b)

Figure 3.5: The a) C code b) our IR language for ‘oneFunc’

We begin by examining the *entry* basic block. The first statement creates the alias relationship between x and y. This causes a gen operation for both of those variables in the Alias set in respect to each other. Statement 2 has a simple constant assignment which involves the statement being assigned to its own blame set. For SSA registers in our calculus, they are always a member of their own blame set. Statement 3 involves a store to non-pointer variable z. We distinguish between stores to non-pointer and pointer variables with different instructions in our calculus, as they are handled in different manners. The store to the non-pointer results in a kill to the current Temp Blame Set (which is empty) and a gen to both the Blame Set and the Temp Blame Set. The next few statements involve the loading of the z value in order to be used in a comparison operation for the conditional branch.

The *bb* basic block has *entry* as a predecessor meaning it takes in as its IN set the OUT set of *entry*. Its first statement, number 7, is our first encounter with pointers. Since $t4$ contains the address of pointer variable x, we gen values in the Up-Pointer and Down-Pointer columns. Statement 9 loads the value of z, and we transfer the blame from the Temp Blame Set of z to the register written, $t6$. We follow that up with a store to a pointer in Statement 10, which calls our *doStore* function. This function follows the up pointer for $t5$ and assigns blame to x. It then goes through all of the aliases of x to assign blame to y as well. Finally, we perform implicit operations on this basic block since it has a conditional relationship with *entry*. Since tB was the register the control flow is based on from *entry*, the blame for that register is propagated to all of the instructions within basic block *bb*.

The *bb*7 basic block has both *bb* and *entry* as predecessors and takes in both

entry:

#	Statement	B	T	U	D	P	F	A
1	store y, $@x$	y:{1} x:{1}						y:{x} x:{y}
2	$t0 = 1$	$t0$:{2}						
3	store $t0, @z$	z:{2, 3}	z:{2, 3}					
4	$t1$ =load$@z$	$t1$:{2:4}						
5	tB =cmp $t1, 0$	tB:{2:5}						
6	br tB, bb, bb7							

bb:

#	Statement	B	T	U	D	P	F	A
7	$t4 = @x$	$t4$:{7}		$t4$:{x}	x:{$t4$}			
8	$t5 = t4 + 4$	$t5$:{7, 8}		$t5$:{x}	x:{$t4, t5$}			
9	$t6$ = load$@z$	$t6$:{2, 3, 9}						
10	store $t6, t5$	x, y:{1:3, 7:10}						
Implicit		x, y:{1:10}						

bb7:

#	Statement	B	T	U	D	P	F	A
11	$t7 = 2$	$t7$:{11}						
12	store $t7, @z$	z:{2, 3, 11, 12}	z:{11, 12}					
13	$t8$ =load $@z$	$t8$:{11:13}						
14	$tB1$ = comp $t8, 0$	$tB1$:{11:14}						
15	br $tB1$, bb12, bb20							

bb12:

#	Statement	B	T	U	D	P	F	A
16	$@(s.i)$ = malloc(4)	$s.i$:{16}						
17	$V \equiv V_1 \rightarrow f$					$s.i$:{s}	s:{$s.i$}	
18	$t9 = @(s.i)$	$t9$:{18}		$t9$:{$s.i$}	$s.i$:{$t9$}			
19	$t10$ =load $@z$	$t10$:{11:12, 19}						
20	store $t10, t9$	$s, s.i$:{11:12, 16:20}						
Implicit		$s, s.i$:{11:20}						

Table 3.3: Applying Blame Calculus to Sample Program

of their OUT sets. The operations in this basic block are simlar to what occurred in *entry*. However, this time the Temp Blame Set for z is not empty so the kill operation deletes statements 2 and 3 from the set. Those statements are still valid in the Blame Set for z. This takes program variable reuse into account. When final blame is reported, variable z will report all of the data writes that occurred during the entire function. However, now if any variable reads the value of z, they will only receive the blame for the latest operations that went into populating the current value of z, and not all the blame for each value z has held over the course of the function.

The final basic block, $bb12$, mirrors the functionality of basic block bb. However, instead of dealing with pointer mappings we are dealing with field mappings. The operation in statement 17 is not seen in the original code, and serves as a way to create a mapping between the parent variable and its field for our calculus. This allows us to perform operations on individual fields with the same set operators that we would use for a standalone variable. The store used in statement 20 is also handled by the *doStore* function. This time, the field mappings cause both s and $s.si$ to mirror each other in terms of allotted blame. This basic block also has a conditional relationship (with register $tB1$ from $bb7$), and the implicit blame is transferred accordingly.

	Statements			Source Lines		
Vars	**Explicit**	**Implicit**	**Total**	**Explicit**	**Implicit**	**Total**
x	1:3,7:10	2:6	1:10	8,9,10,12	10,11	8:12
y	1:3,7:10	2:6	1:10	8,9,10,12	10,11	8:12
z	2,3,11,12		2,3,11,12	8,10,14		8,10,14
s	11,12,16:20	11:15	11:20	7,14,16,17	14,15	7,14:17
s.i	11,12,16:20	11:15	11:20	7,14,16,17	14,15	7,14:17

Table 3.4: Statements & Source Lines Attributed to Variables at End of Function

3.1.4 Interprocedural Formalization and Transfer Functions

The dataflow interactions modeled in our calculus need to be propagated across function calls. When looking at the functions involved for the call trace at each sample, we propagate blame through *transfer functions*. Based on the sets of blamed parameters for each function call, the transfer functions propagate the sets we had previously calculated through intraprocedural analysis. Transfer functions are discussed in further detail in Section 3.2.2.

We account for side effects in a similar way with a modified transfer function. Side effects are discussed in further detail in Section 3.3.1.3. The formalization for the transfer functions and side effects is shown in Table 3.5. The sets utilized include the following:

Blame (B)

The same Blame Set that is utilized in intraprocedural analysis for each register and variable.

Transfer Function Blamed Parameters (BP)

The blamed parameters are determined at runtime based on blame analysis

on the callee function. This information is propagated through a transfer function to determine which parameters in the caller function are the blamed parameters. There is one instance of this set per call.

Transfer Function Non-Blamed Parameters (NBP)

The non-blamed parameters are those parameters that are determined at run-time to not be blamed in the callee function. The intersection of the non-blamed parameters and blamed parameters is the null set and the union is all of the parameters for the callee function. There is one instance of this set per call.

Side Effect Blamed Parameters (SBP)

Since the side effects occur in a function that a sample did not generate an interrupt in, there is no true parameter that can be blamed like in the standard function. Instead, we look for relationships between the parameters in the side effect function. If there is a pairwise blame relationship from one parameter to another within that function, the blamed parameter is placed in this set. There is one instance of this set per call.

Side Effect Non-Blamed Parameters (SNBP)

The parameters that place blame on another parameter passed into a side effect function. There is one instance of this set for each member of the set SBP.

	Blame (B)	**Aliases (A)**
Statements	**Gen**	**Gen**
$r = \text{call}(v1, ...)$ (Transfer Function)	$\forall a$ in BP($call$) $\forall b$ in NBP($call$) a : { B(b)} r : {r, B(b)}	$\forall \{a, b\}$ in Aliases($call$) A(a) : {b} A(b) : {a}
$r = \text{call}(v1, ...)$ (Side Effect)	$\forall a$ in SBP($call$) $\forall b$ in SNBP(b) a : { B(b)}	$\forall \{a, b\}$ in Aliases($call$) A(a) : {b} A(b) : {a}

Table 3.5: Interprocedural Gen Kill Sets

Aliases (A)

Alias relations are determined in the same manner as for intrapocedural analysis. However, they are limited to alias relationships between parameters to the functions. There is a set of each pair of aliases for each call. This is in addition to the individual set of aliases stored by each register and variable.

3.2 Blame Calculation

We use the LLVM [39] IR (intermediate representation) to instantiate the concepts introduced in the formalization. LLVM can process C,C++, and FORTRAN source code. LLVM follows the SSA convention for its registers. LLVM represents variables from the original compiled code as memory locations that LLVM registers move data to and from through load and store operations.

Table 3.6 shows the LLVM instructions that correspond to the code in Figure 3.1 and the blame sets generated at every step based on our defined gen-kill sets.

3.2.1 Graph Representation

The gen-kill sets (and corresponding in-out sets) are stored per register. Because of the chained nature of dataflow interactions in the program, this can lead to redundant data from set to set. We eliminate this redundancy by representing the dataflow interactions as edges within a graph. This way the blame calculations can be represented as graph traversal problems and the blame sets need only be explicitly stored in a subset of the vertices, such as those corresponding to variables declared in the target program. The graphical representation of the LLVM snippet and its dataflow is shown in Figure 3.6. The variables from the original, compiled program are represented in pink (shaded) boxes. The SSA registers generated by LLVM and the constant values are the vertices represented by white ovals. The LLVM instruction types are the labels for each edge.

Operations on the graph representation emulate the same operations as the gen-kill sets. For 'store' operations, registers in the intermediate representation are added to the blame set for variables from the profiled program. Implicit operations are represented by directed edges going from every register written in the affected

LLVM Instruction	**Blame Sets Modified**
store i32 5, i32* %a	a:B{C:5} a:T{C:5}
store i32 6, i32* %b	b:B{C:6} b:T{C:6}
%tmp1 = load i32* %a	tmp1:B{tmp1, C:5}
%tmp2 = load i32* %b	tmp2:B{tmp2, C:6}
%tmp3 = add i32 %tmp1, %tmp2	tmp3:B{tmp1, tmp2, tmp3, C:5, C:6}
store i32 %tmp3, i32* %c	c:B{tmp1, tmp2, tmp3, C:5,C:6}

Table 3.6: LLVM representation and Blame Sets for snippet from Figure 3.1

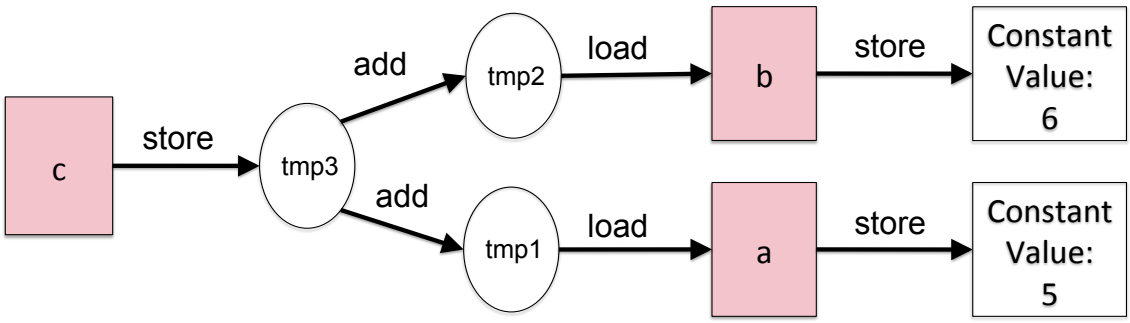


Figure 3.6: Internal LLVM Graph Representation of snippet from Figure 3.1

.

basic block to the conditional register. In our graph, these edges are represented by grey, dashed edges.

The most important instruction of the LLVM IR that does not have a corresponding element in our formalization language is the 'getelementptr' (GEP) instruction [42]. The GEP instruction gets the address of a complex data structure or array but does not actually access memory. A load instruction usually follows a GEP instruction to get the value from the calculated address. We do back traversals along GEP and load operations to resolve the pointer relationships (in the same way as outlined by our doStore function in Figure 3.2). In our graphs, the GEP instruction is represented by "BASE" or "FIELD" depending on if the GEP instruction was working with standard pointers or was indexing fields within a complex data type.

In many cases, we can compress the graph by storing only the line numbers for certain operations rather than all the registers (or constant values) that feed into the variable. The reason for this is based on how we use our blame mappings. At runtime, based on the debugging information and the samples that occur we

can only resolve the sample to a line number within the program. Because the line number is the most accurate indicator we can achieve with the runtime data, we do not gain any accuracy in our measurements by storing the operations for our static analysis at any level more detailed than line number.

However, since blame is propagated through writes, this does not affect our analysis in most cases. In most of the code we examined, one statement (or write) per line was the normal practice. If we were to encounter code that had multiple statements per line, we could add a compilation phase step that put every statement on a different "source line," while storing the reverse mappings so our blame information could still be utilized. However, that would involve a recompilation of the original program.

An alternative is to perform blame analysis on assembly code. This would achieve instruction level preciseness, but this approach has many challenges. We discuss this option further as future work in Section 6.2.

We can compress the graph where we have redundant information until we achieve a truncated representation. The compact representation for the graph shown in Figure 3.6 is shown in Figure 3.7. This graph contains only the variables from the original program and the blame relationship between them (the directed edges with the 'B' label). Each node in the graph also contains additional information about what source line that operation maps to. This allows us to map the low level data flow operations in the intermediate representation back to the original source code. The applicable line numbers blamed for each variable are shown in brackets for the nodes in the compact representation.

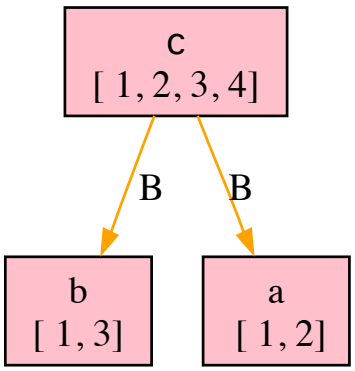


Figure 3.7: Compact Representation of the graph from Figure 3.6

The vertices that are moved from the raw graphical representation to the compact graph meet a certain criteria to be migrated. All local variables and function parameters are migrated automatically. Fields that are read or written also migrate. Registers may migrate if they meet special criteria. Any register that is a pointer to one of the elements listed are eligible. However, if there are multiple registers that are used to calculate an address only the most relevant is included. For instance, we may have one register that contains a pointer and another register that contains an offset to that pointer. We can eliminate the first pointer if its only access is to contribute to the calculation of the second register. We also migrate any register that is a parameter to a function call. Finally, we migrate any register that is the value transferred in a store operation. This allows us to simulate our Temp Blame Set graphically, by having a directed edge to that register instead of the non-pointer local variable.

Node Type	Shape
Local Variable	Rectangle
Parameter	Inverted Triangle
Register (pointer)	Diamond
Register (non-ptr)	Oval
Field	Parralelogram

a)

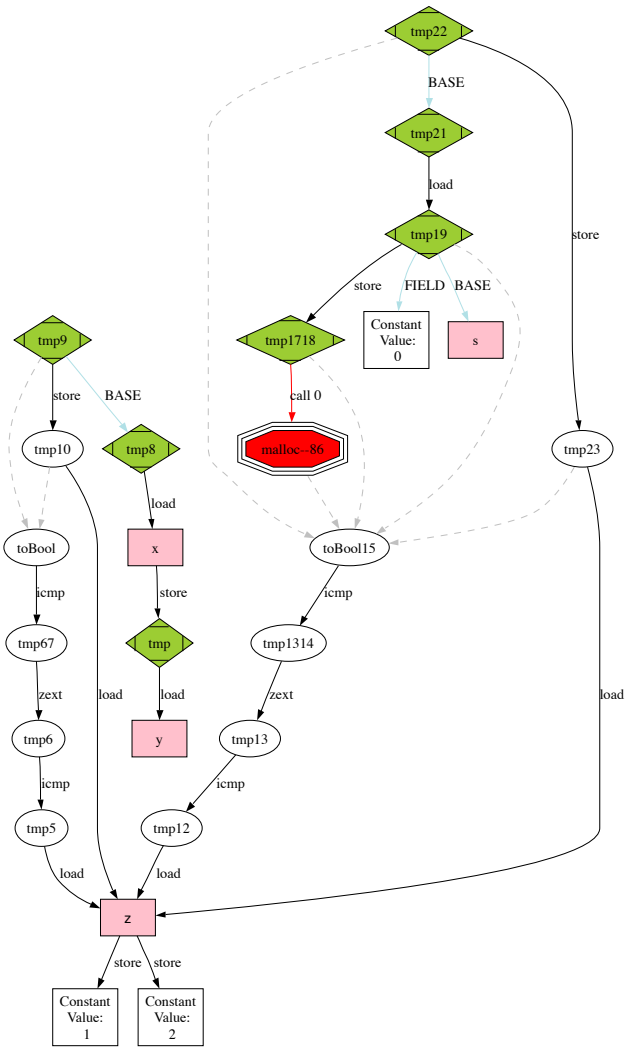
Edge ID	Closest Set Equivalent
B	Blame Set
D	Down Pointer
F	Fields
S	Temp Blame Set
A	Alias

b)

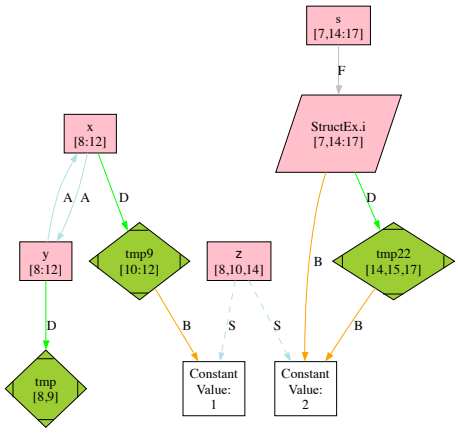
Table 3.7: Graph key for a) Nodes and b) Edges

Once we have established which vertices will be migrated, we do a separate recursive traversal through the graph for each migrated node. When we reach another migrated node, we create a new edge between the two based on their relationship and return. The set of line numbers for all non-migrated nodes encountered on the traversal are stored by the migrated node.

The raw and compact graph representation for the example function we examined earlier, 'oneFunc' is shown in Figure 3.8. Table 3.7 provides a key to the types of vertices and edges in the graphs. Those registers that are eligible to be migrated are shown in green (shaded). By comparing the two graphs, you can see that not all eligible nodes get migrated. This is due to multiple registers being utilized to calculate pointers. The raw graph, with vertices for every SSA register generated by LLVM, lines up closer to the operations that were performed by our blame calculus. The compact graph represents a higher level view of the blame calculations and relationships that are occurring within the program.



a)



b)

Figure 3.8: The a) raw graph and b) compact graph for ‘oneFunc’

3.2.2 Transfer Functions and Interprocedural Blame

The data flow relationships we have discussed so far were recorded at the function level and calculated prior to program execution through intraprocedural analysis. For interprocedural analysis, we need a mechanism to communicate the blame between functions. We utilize our transfer functions for this step. When looking at the data flow graph, we utilize a form of escape analysis [19] to determine for each function the set of variables, which we call exit variables, that are live outside of the scope of the function. These could be parameters passed into the function, global variables, or return values. All explicit and implicit blame for each function's transfer function is represented in terms of these exit variables during static (pre-execution) analysis.

We use a graphical representation for our interprocedural analysis as well. For callee functions, we look at the vertex representing each exit variable and determine whether that exit variable is blamed for that sample point at runtime. For caller functions, we have multiple parameters that have incoming edges to a node representing a function call. At runtime, we use the transfer function to match the blamed exit variable(s) from the callee to the blamed parameter(s) in the caller. Once we have that information, we can reorder the edges of the graph such that the blamed parameters have outgoing directed edges to each parameter that is not blamed.

Figure 3.9 shows source code, LLVM intermediate representation, and corresponding compact internal blame representation for a small snippet of code that

performs a call to function *foo*. At runtime, we can determine how the call to *foo* (the callee) affects the data flow relations within our caller function. With information from *foo*, we have four possible blame relationships for the parameters detailed in Figure 3.10. Based on which parameter(s) are blamed, we manipulate the dataflow graph in the caller function and then perform our blame calculations on our graph accordingly.

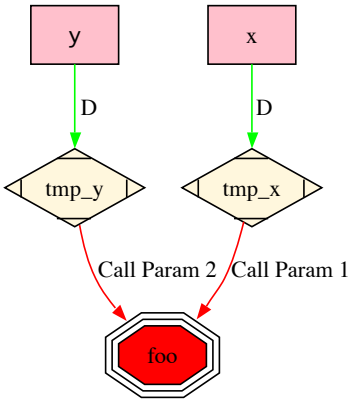
Because of the potential of graph manipulations for transfer functions, in actuality there are always two versions of our blame graph. The first version is the base graph that we initially generate based on the intraprocedural analysis. This base graph has the call nodes represented with input parameters, but performs no deeper interprocedural analysis. The second graph is the graph with the manipulations displayed in Figure 3.10. The graph is manipulated based on the results of the transfer function and information we have gathered from the callee. The graph manipulation changes the blame relationships, and this graph is the one we use to assign blame to the variables. When the sample has been processed, the graph reverts back to its base state and awaits the next sample point.

3.3 A Blame Tool

Variable blame is fundamentally a technique for program understanding and performance analysis. We have discussed the formalization of variable blame and its graphical representation, but did not go into detail about the components needed for a tool that could utilize variable blame.

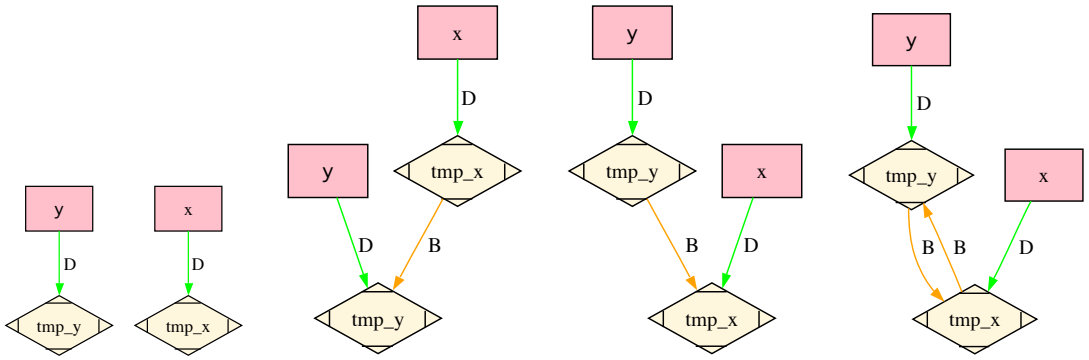
```
int *x, * y;
...
foo(x,y);
```

```
%y = alloca i32*
%x = alloca i32*
...
%tmp_x = load i32** %x
%tmp_y = load i32** %y
call i32 @foo(i32* %tmp_x,
      i32* %tmp_y)
```



a) b) c)

Figure 3.9: a) C code, b) LLVM Intermediate Representation, and c) Internal blame representation for snippet that uses a transfer function



(a) None Blamed (b) X Blamed (c) Y Blamed (d) X and Y Blamed

Figure 3.10: Transfer function graph transformations based on parameter(s) blamed

This section discusses the construction of a blame centric tool. An important factor for our analysis was minimizing program perturbation. For this reason, we designed our system to push as much processing as possible to static analysis (pre-run and post-mortem). At runtime, we utilize sampling instead of direct measurement through caliper based instrumentation. Figure 3.11 shows the components of the tool we built for determining blame. Step 1 consists of all intraprocedural analysis and can be run on a single processor. Step 2 is the actual running of the program

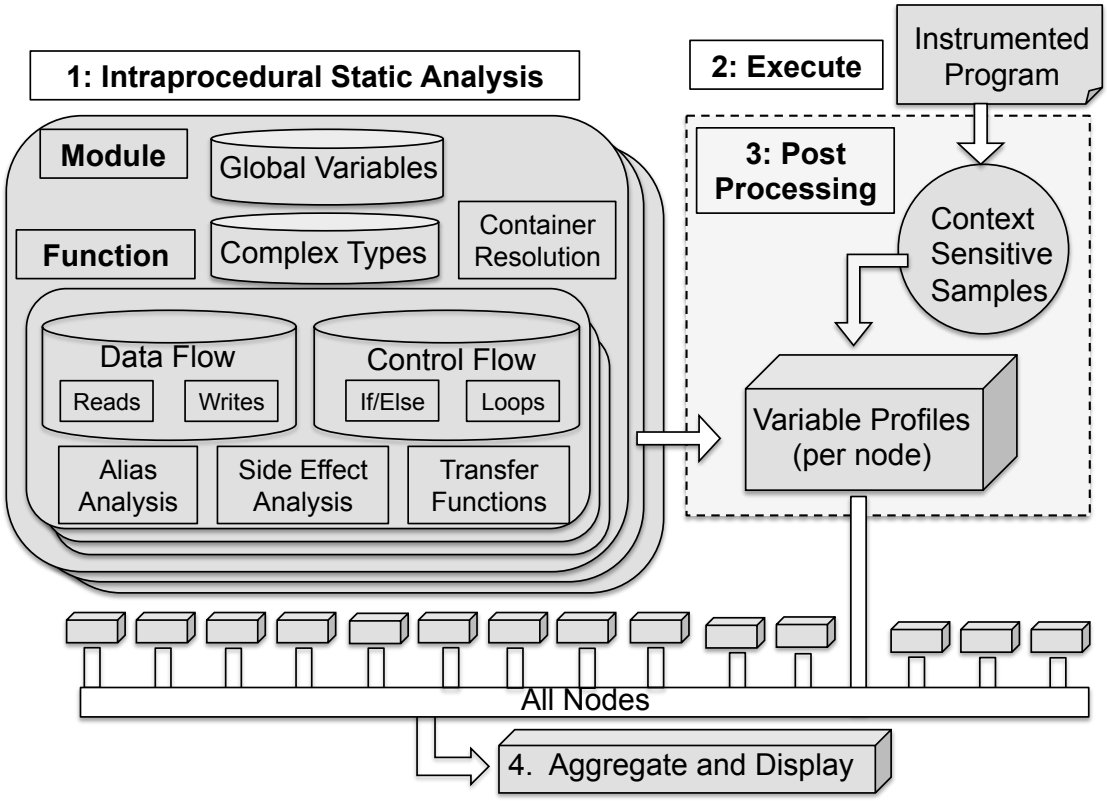


Figure 3.11: Blame Tool Components

that has been modified to enable event driven sampling. Step 3 is an embarassingly parallel post processing step, but could be run on a single processor. Step 4 involves the presentation of the aggregated data via a GUI. This section will describe each of these steps in further detail.

3.3.1 Intraprocedural Static Analysis

Although our complete analysis is interprocedural, we limit the analysis to intraprocedural at this point for two reasons. First, we need runtime information for interprocedural data flow relations and alias analysis. Some interprocedural analysis could be performed before the program is run, but it would be incomplete.

Secondly, by limiting static analysis to intraprocedural information we can easily reuse analyses from run to run (or among programs that use shared libraries). For example, in libraries such as PETSc [5, 6] the code will be used by multiple applications. Performance profiling is an iterative process where code may be modified based on values given by the tool. We store analysis at the procedure level, so the analysis need only be rerun on those procedures that are modified.

We perform intraprocedural analysis to create transfer functions, to perform container resolution for complex data types, for alias analysis, and to perform side effect analysis. Each one of these analyses is augmented with runtime information, but we discuss each analysis here because the majority of the analysis is stored at the procedure level.

3.3.1.1 Transfer Functions

We discussed transfer functions previously in Sections 3.2.2 and 3.1.4. For each procedure, we export a list of exit variables, local variables, and the elements of the graph that map up to them in an ancestor relationship. The exported graph is the minimal representation with redundant data eliminated. To resolve the transfer function, we determine post-mortem which variables are blamed for each sample point. This is determined by analyzing if a blamed vertex is a descendant of a given exit or local variable. This section will discuss special cases that transfer functions need to address.

One special case that requires manual intervention is when source code is not

available. The common case for this is when dealing with library functions and system calls. For these cases, transfer functions can be created based on knowledge about the functionality. A small case study of this can be found with a common library, "math.h." Every function within that library (sin, cos, tan, ...) has a prototype that takes in all scalars as parameters and returns a scalar. Since none of the parameters are pointers or pass by reference if we make a small assumption that the function does not significantly utilize global variables, we can safely claim that all the blame for the function lies within the return value. Therefore, any call to one of these functions will transfer all blame to the variable that stores the return value for the function.

When faced with a complete lack of knowledge about about a function (no source or documentation) a series of heuristics are used that divides up the blame between the parameters and return values from these functions. These heuristics deal primarily with the function prototype for the function. In cases where there is a return value and void parameters, we assign all blame to the return value. In cases where pointers are passed in as parameters, we split blame evenly among the pointers to the function.

Another special case is when dealing with function pointers. We are able to handle this case with our base implementation. Because our pre-run analysis is intraprocedural, we mark a call using a function pointer in the exact same manner as any other call and no further analysis is needed. At runtime, we gather the stack traces and can resolve the function pointer. At that point, we can apply the transfer function in the same way we would with any standard call.

A final special case to address is the way transfer functions deal with recursion. In the case where the recursive function is repeatedly called from the same source line in frame after frame, the transfer function is reapplied at each frame as we progress up the call stack. We are able to do this because the transfer function would resolve to the same blame relationships because the inputs would be the same from function to function. However, if the recursive function is called from a different source line from frame to frame we recompute the transfer function for each caller function. This is because the blame results differ based on the point in the function where a call occurs.

3.3.1.2 Container Resolution

Container resolution refers to the resolution of blame within complex data types. For instance, a structure or class may have multiple fields which acquires blame through the course of the program. Some of these fields may themselves be classes or pointers to classes. Container resolution is simply the bubbling up of blame until it reaches the top most container type. Much of this resolution can be taken care of statically, though there could be cases where a field may be referenced through a set of pointers (such as void*) where runtime information will be needed to fully attribute blame to the proper container. Container resolution is the core piece of analysis that allows blame to be an inclusive profiling method.

```
void doStuffOnSide(int * x,  int * y)
{
    for (int i = 0; i < N; i++)
          x[i] = y[i];            // SAMPLE POINT 1 (No side effect)
}

void baz(int * x, int * y)
{
     doStuffOnSide(x,y);
     x[0] = 0;                   // SAMPLE POINT 2  (Side effect)
}

void main()
{
   ...
   baz(x,y);
}
```

Figure 3.12: Sample Side Effect

3.3.1.3 Alias and Side Effect Analysis

We discuss alias and side effect analysis together because they both rely on similar interprocedural components. There is also some overlap since some of the alias relations are also side effects themselves. Alias analysis at the procedure level is performed using the same graphs we use to perform blame analysis. The GEP instruction presents explicit representations of memory locations and we create points-to sets based on these locations.

Side effect analysis is important for our blame analysis because our data gathering is sampling based. For that reason, we have to account for what occurs in those function calls that are not in the calling context of the actual sample. An example of side effects we have to account for is presented in Figure 3.12. The first sample point has no side effect associated with it. The sample occurs with the con-

```
void absorbEV(int * x,  CStruct * s)
{
    s->x = x;
}

void doStuff(CStruct * s)
{
   s->x[0] = 1;
}

void top(int * x)
{
     CStruct s;
     absorbEV(x, &s);
     doStuff(&s);
}
```

Figure 3.13: Sample Side Effect Alias

text *main→baz→doStuffOnSide*. At the point of the sample there are no function calls that could affect the blame relationships for that sample. However, for the second point the context for the sample is *main→baz*. There is a possibility that the call to the *doStuffOnSide* function on the line before the sample could cause a side effect, specifically in the blame relationship between x and y. For these cases, we perform special transfer functions for the side effect calls based on the sample point. In this example, regardless of control flow, x would get the blame for the operations on y. In cases where control flow would dictate which parameter would get the blame (such as in the *bar* function from Figure 3.14), we assign blame to all possible parameters for side effect functions.

An example showing side effects and alias analysis together is shown in Figure 3.13. This example is especially relevant for our analysis because of the potential to mask an exit variable in a transfer function. The function *top* has one exit variable,

a pointer to the integer x. This would be the variable passed through a transfer function. The function *top* makes a call to *absorbEV* which creates an alias to a field within a structure. The next function called from *top*, *doStuff* now only passes in that structure instead of the original pointer. This style of aliasing is very common with work vectors in scientific applications. These vectors are defined locally within a function and then are aliased to fields within large derived types that are passed in as parameters or defined globally. At runtime, we examine the potential aliases from side effects based on the sample points. We resolve these aliases to attribute values correctly to locally declared variables that are complex types and to make sure there is not a break in our chain of transfer functions.

3.3.2 Execution

The execution step is run on a program that has been modified to enable event driven sampling.

3.3.2.1 Instrumentation

Instrumentation is accomplished by modifying a binary program (or source) to add code at the beginning of program execution to trigger periodic sampling and to record results at the end of the program. The resulting binary is also linked with a shared library that contains the per sample handler routine which performs the stack walk to get the full call path for each sample. Since the instrumentation is done on the binary and the new executable is rewritten, no source changes need to

be made and the program need not be re-compiled. We accomplish this by using the Dyninst binary rewriter tool. When profiling programs on architectures where Dyninst is not supported, we perform standard source instrumentation.

We enable sampling by using the hardware counters and overflow interrupts provided by PAPI [11]. We are able to generate samples based on any of the available hardware counters for the architecture where the profiled program is running. At each interrupt point, we are given the thread context which includes the register state at the time of sample. We use this register state (mainly the registers for the program counter, frame pointer, and stack pointer) to get the full calling trace by performing a stack walk at the sample point. The stack walking for each sampling point is performed using the StackWalker API [62].

3.3.2.2 Running the Program

The modified binary is run exactly the same way as the original. By pushing most of the analysis to pre-run, and by using sampling, we can minimize program perturbation relative to that of a traditional context sensitive profiling approach. A file is output for each thread containing raw addresses of the sampled instruction at every level of the call trace for each sample.

3.3.3 Post Processing (per thread)

The final step is to take the raw context sensitive samples and the stored intraprocedural analysis to determine the final blame for each thread. After resolving

the addresses to functions and line numbers, we can use the predetermined exit variables to apply transfer functions at each level of the call trace. This means we can bubble blame information up as far as we need to assign it to the appropriate complex data type. For local variables, blame can be assigned without the use of transfer functions. The percentage blame for any given variable is the number of samples attributed to the variable divided by the total number of samples.

3.4 A Small Example

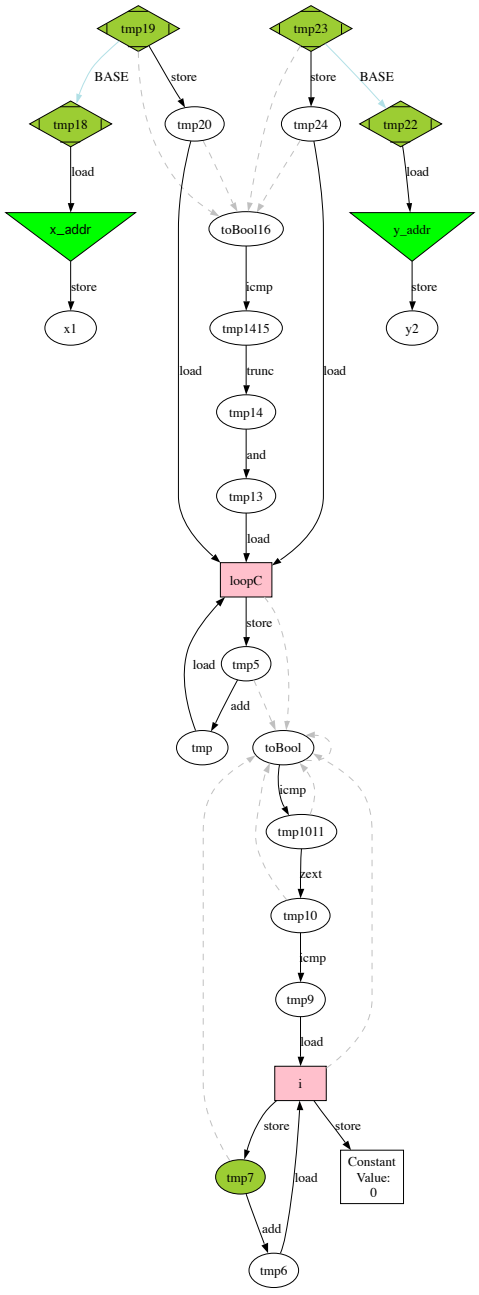
To make the calculation more concrete, we now show the steps of calculating blame using a small sequential program (shown in Figure 3.14).

3.4.1 Intraprocedural Analysis

We first examine the *bar* function. The LLVM IR and its graphical representation are shown in Figure 3.15. The exact LLVM instructions have been slightly modified to make them more readable for this document. The original LLVM instructions for *foo* and *bar* can be viewed in Appendix A. The exit variables, x_addr and y_addr are represented as green inverted triangles. Registers that are pointers are shown as diamonds. This function highlights implicit operations (both loops and conditionals) and the handling of GEP instructions in terms of array indices. The GEP operation provides a pointer for the index into both arrays (in this case 0). This pointer is the target of the store. The implicit operations are highlighted by dashed directed edges. We see implicit edges from the x and y array indices

```
1  typedef struct {
2  int * sX; int * sY;
3  } StructEx;
4
5   void bar(int *x, int *y)
6   {
7    int i, loopC = 0;
8    for (i = 0; i < N; i++)
9       loopC++;     // SAMPLE POINT
10
11    if (loopC %2)
12      { x[0] = loopC;} //SAMPLE POINT
13    else
14      { y[0] = loopC;} //SAMPLE POINT
15  }
16
17  void foo()
18  {
19    StructEx se;
20    se.sX =(int*)malloc(N*sizeof(int));
21    se.sY =(int*)malloc(N*sizeof(int));
22    bar(se.sX,se.sY);
23  }
24
25  void main()
26  {
27    foo();
28  }
```

Figure 3.14: Sample Blame Program



a)

```
define void @bar(i32* %x1, i32* %y2) {
entry:
// LINE NUMBER 8
      store 0, i
      br label bb8

bb:
// LINE NUMBER 9
      tmp = load loopC
      tmp5 = tmp + 1
      store tmp5, loopC
// LINE NUMBER 8
      tmp6 = load i
      tmp7 = tmp6 + 1
      store i32 tmp7, i
      br bb8

bb8:
// LINE NUMBER 8
      tmp9 = load i
      tmp10 = compare tmp9, 9
      tmp1011 = cast tmp10
      toBool = compare tmp1011, 0
      br toBool, bb, bb12
      br bb12

bb12:
// LINE NUMBER 11
      tmp13 = loopC
      tmp14 = and tmp13, 1
      tmp1415 = cast tmp14
      toBool16 = compare tmp1415, 0
      br toBool16, bb17, bb21

bb17:
// LINE NUMBER 12
      tmp18 = load x_addr
      tmp19 = tmp18 + 0
      tmp20 = load loopC
      store i32 tmp20, tmp19
      br return

bb21:
// LINE NUMBER 14
      tmp22 = load y_addr
      tmp23 = tmp22 + 0
      tmp24 = load loopC
      store tmp24, tmp23
      br return

return:
      ret void
}
```

b)

Figure 3.15: The a) internal graph representation of 'bar' function and b) LLVM IR

to the load from *loopC* (specifically the branch register, *toBool*16, that is blamed for that load), since both of these writes are based on conditional operations on that variable. We also see an implicit operation between *loopC* and the load for i,

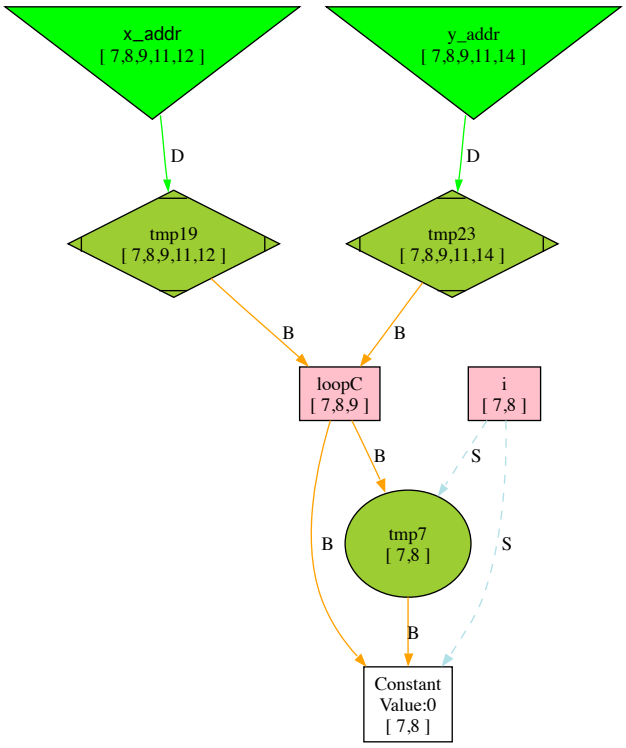
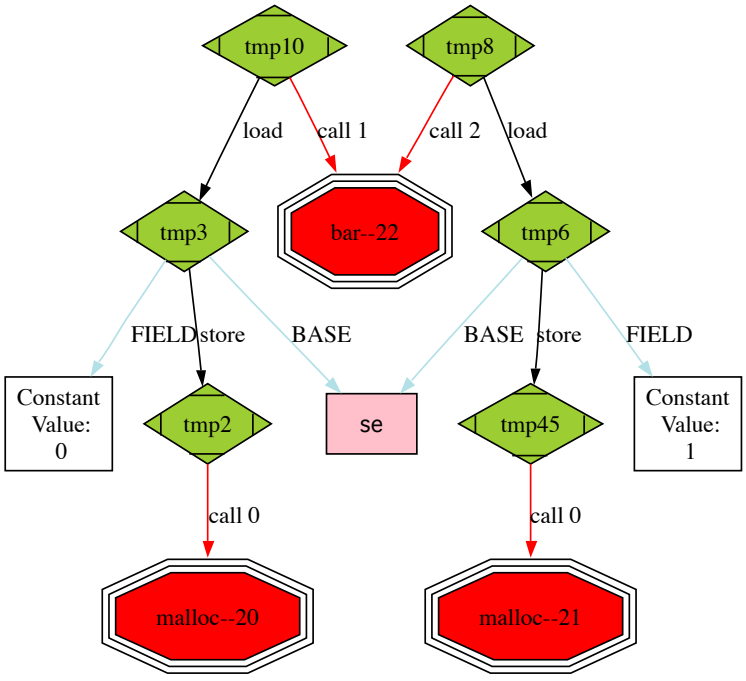


Figure 3.16: Final Internal Representation of 'bar' function

because i is the loop iterator for the loop that increments the value of *loopC*.

The compact graph representation of the *bar* function is shown in Figure 3.16. The down-pointer relationship with the index into their respective arrays is represented with a directed edge to the pointer to the address with the 'D' label. There are blame edges between the indices and the *loopC* variable, but there is no direct blame edge from *loopC* to i, though we have established there is an implicit relationship between the two. This is due to the fact that i has multiple data writes and the edges are to each of the sources for the 'store' operations (represented by the dashed edge and the 'S' label), and not the variable itself.

The operations in the *foo* function details how we internally represent de-



a)

```
define void @foo() {
entry:
// LINE NUMBER 20
      tmp2 = call @malloc( 40)
      tmp3 = ptr to address of first field of se
      store tmp2, tmp3
// LINE NUMBER 21
      tmp45 = call @malloc( 40 )
      tmp6 = ptr to address of second field of se
      store i32 tmp45, tmp6
// LINE NUMBER 22
      tmp8 = load tmp6
      tmp10 = load tmp3
      call @bar( tmp10, tmp8 )
      ret void
}
```

b)

Figure 3.17: The a) LLVM IR and b) internal graph representation of 'foo' function

rived types, and also shows how function calls are represented. The LLVM IR and its graphical representation are shown in Figure 3.17. We can see that the GEP instruction is used to access fields from a derived type.

The truncated version of the graph is shown in Figure 3.18. The 'F' edges

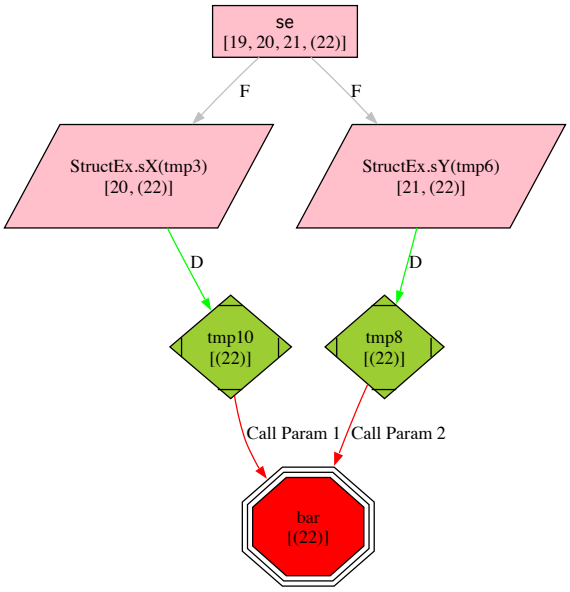


Figure 3.18: Final Internal Representation of 'foo' function

represent a "field" relation between a variable and its fields. We also see how the fields are passed in as parameters to the *bar* function. After the program is run, we can determine which parameter is to blame and readjust the graph accordingly.

Name	**Function**	**Local/Exit**	**Blame Lines**
i	bar	Local	7, 8
loopC	bar	Local	7, 8, 9
x	bar	Exit	7,8,9,11,12
y	bar	Exit	7,8,9,11,14
se	foo	Local	19,20,21,(22)
se.sX	foo	Local	20, (22)
se.sY	foo	Local	21, (22)

Table 3.8: Blamed source lines for each local variable

With the final internal graph representations of both *foo* and *bar*, we can determine the blamed source lines for each program variable in our program. This is represented in Table 3.8. We place line number 22 in parentheses. This is because

depending on the outcome of the transfer function, this may be attributed to either of the fields of *se*. The resolution of the transfer function depends on runtime information, specifically where the sample occurs within the callee function and how blame is calculated in the callee as a result. Since our analysis is inclusive, the parent variable *se* gets the union of the blame for all of its fields. For this reason, it has both lines 20 and 21 attributed to it.

3.4.2 Runtime Sampling

In this example we will assume that when the program is run, three samples occur. Each sample is marked by "SAMPLE POINT" in the code depicted in Figure 3.14. The sample points and their context are shown in Table 3.9.

Sample	**Context**	**Line Number**
1	main→foo→bar	9
2	main→foo→bar	12
3	main→foo→bar	14

Table 3.9: Line numbers and context for samples

We use the runtime data to attribute the samples to the appropriate variables. Sample 1 occurs calculating the local variable *loopC*, and blame is attributed to that variable. Furthermore, the calculation of *loopC* falls within the blame set of x and y, so they are assigned blame as well. For any given sample, multiple variables may share blame. In a given function, the total percentage assigned to all variables may be more than 100% for this reason. Sample 2 occurs on a direct write to x, and x is the sole variable blamed. Sample 3 is a write to y, and y is blamed.

Name	Function	Blame %
se	foo	100%
se.sX	foo	67%
se.sY	foo	67%
loopC	bar	33%

Table 3.10: Blame Percentages for Sample Program

3.4.3 Post-Mortem Interprocedural Analysis

After execution, a post-mortem processing is performed to combine the sampled data and the static analysis. The transfer function is then applied to each sample. We resolve through the transfer function that the exit variable x from *bar* corresponds to $se.sX$ and y applies to $se.sY$. We also see that foo has no exit variables so no further transfer function needs to be applied. se is the container variable for both of the fields so it also gets blame attributed to it. The final blame percentages for all the local variables in the program are shown in Table 3.10. All percentages are in terms of the entire program, not just local to the function the variable is defined in. Recall, exit variables are only intermediate steps to bubble blame information up, so we do not list them.

3.5 Data Presentation

At this point we will have blame mappings from performance data to program variables, but still at a per process level. For parallel and distributed programs, we aggregate the results across the processes for ease in presentation while still maintaining the drill down information for single processes. This information can

be used to identify load imbalances associated with populating the values of the data structure. This section discusses how we aggregate and present the data. A screenshot of the components of the GUI with all of the elements is shown in Figure 3.19.

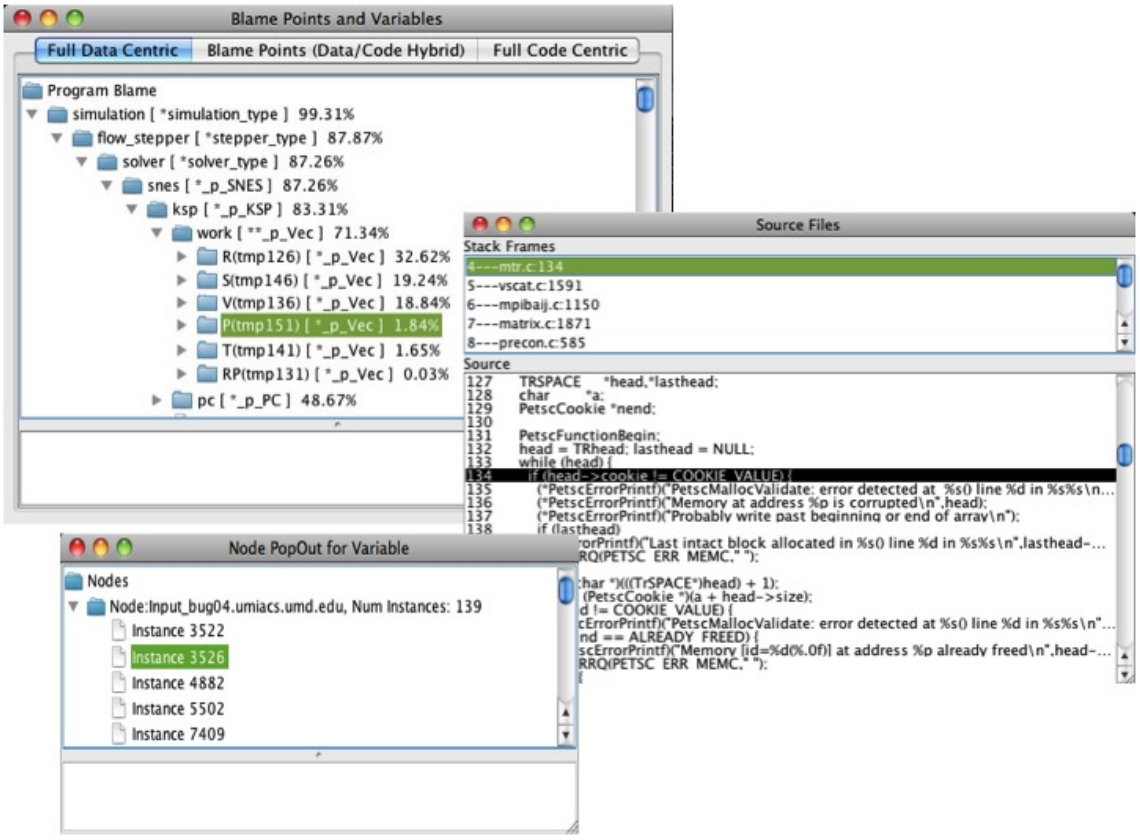


Figure 3.19: GUI Screenshot

3.5.1 Main Display Categories

There are three ways to view the data: a flat data centric view, a traditional code centric view, and a hybrid approach. Figure 3.20 shows each of these views with the data from the example in Section 3.4.

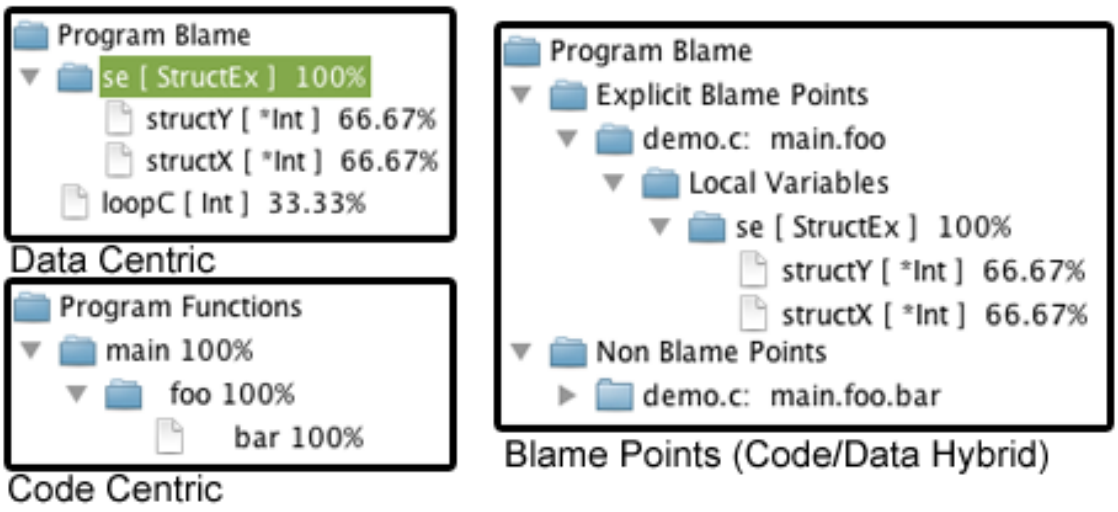


Figure 3.20: Main Display Categories

3.5.1.1 Flat Data Centric

The default view we provide to the user is a flat view of all of the variables defined in the program, ranked in descending order by the percentage blame they are assigned. We also present an interface to filter what data types the user sees. This is especially useful when the user is only interested in derived types or certain abstractions (such as a specific sparse matrix or solver in a numeric library).

3.5.1.2 Blame Points

The second view is a hybrid approach between code and data centric using "blame points." Blame points are points in the program that are deemed to have interesting variables. These points can either be explicit or implicit.

Implicit blame points are automatically defined by the tool whenever there are variables that can not be bubbled up any further in the call stack. This occurs in the case where the set of exit variables for a function is null (i.e. a void function

with no parameters and no global variables accessed).

Explicit blame points can be assigned using whatever criteria the user decides. An example heuristic would be to assign an explicit blame point to any function containing a variable that has over 50% of the program's blame.

Once the blame points are determined, they are presented to the user as a list of the functions for each type of blame point with the opportunity to examine the local variables for each.

3.5.1.3 Code Hierarchical

We also include a traditional code view. We maintain that blame augments code centric views, not supplants them. Furthermore, since we have already obtained the context sensitive samples, a code hierarchical view is something we can present with no additional overhead.

3.5.2 Secondary Displays

For each view the user has the opportunity to drill down and look at the samples that make up the percentage blame for a variable. For each sample, the full stack trace can be viewed with each frame having the corresponding source file viewable. Each primary view also has metadata information within the window. In the case of the variable view, it contains information about where the variable was defined, and statistical values (max, min, median) from the threads (the mean is the blame percentage displayed in the primary view).

Figure 3.19 shows the GUI with the flat, data centric main display and utilizing all of the secondary displays discussed in this section. The screenshot is from PFLOTRAN [52], which we will discuss in detail in Section 4.3.3. The hierarchical view in the window on the left is from the primary view and represents the data structure for the blamed variable. Each new level represents a progression through the fields of the type. Starting from the top, *simulation* is the name of the variable. *simulation* has a field called *tran_stepper* which has a field called *solver*. The data types, which we can choose to filter by, are in brackets next to the name of the variable/field. Double clicking on a variable name drills down to the view displayed in the other two windows. The right window gives the complete stack trace for that sample and the source view of the exact line number for the point that triggered the sample for that level in the call stack.

3.6 Summary

In this chapter we have formally defined variable blame and given a calculus to show how it can be calculated using gen-kill sets. We discussed how we optimize space by representing the dataflow, registers, and variables as a compact graph. We discussed the blame tool we created using these representations and discussed how blame data can be represented to the user. Finally, we showed how to calculate blame for a small example program.

Chapter 4

Variable Blame Experiments

Our set of experiments were designed to cover a breadth of program types and show how blame can be used to improve program understanding and performance. This chapter also reports the overhead of computing variable blame at all phases in respect to the profiled program (pre-execution, execution, and post-execution). We also introduce a metric to determine how unique the results given by blame are in comparison to established code centric techniques. This metric is then applied to our case studies along with the blame data.

4.1 Preliminary Experimental Results

To show how our mapping differs from traditional techniques, we chose two small programs that directly exhibit properties that would be found in large parallel programming abstractions. For both programs, the blame metric is computed based on sampling triggered every predetermined number of cycles. For the first experiment, we present the absolute blame numbers that are matched one to one with the samples taken while profiling the program. For both programs, we present the percentage of the program cycles that were used in the calculation of the variable according to our blame mappings. For these preliminary experiments, variable blame is divided equally for each sample across every variable blamed for that sam-

ple. For that reason, the sum of the percentages of the variables will add up to 100%. This is not the case for the later experiments presented in this chapter and for our current blame tool. Under our current implementation, blame is not divided among the blamed variables for a sample.

For both experiments, we used the blame mappings derived from LLVM and real sample points generated by using the PAPI framework [11]. After running the experiments with our tool, we manually inspected the code to verify our blame analysis was reporting the correct information in regards to our defined blame calculus.

4.1.1 FFP_SPARSE

One of the test programs we examined was FFP_SPARSE [21], a small open source C++ program that uses sparse matrices and a triangle mesh to solve a form of Poisson's equation. It consists of approximately 6,700 lines of code and 63 functions. Although this program is sequential, the problem space and data structures utilized are typical of parallel scientific programs and thus make it an attractive case study.

We ran the FFP_SPARSE program and recorded 101 samples which are the basis of the mappings discussed in this section. After removal of the debugging output, the only blame point for this program is the main function, with the program culminating in the output of the final solution vector.

This program does not have complex data structures to represent vectors and matrices, but the variable names for the scalar arrays map nicely to their mathematical counterparts in many cases. Table 4.1 shows the blame mappings for the

Name	**Type**	**Description**	**Base Data Centric**	**Blame(%)**
node_u	double *	Solution Vector	0	35(34.7)
a	double *	Coefficient Matrix	0	24.5(24.3)
ia	int *	Non-zero row indices of *a*	1	5(5.0)
ja	int *	Non-zero column indices of *a*	1	5(5.0)
element_neighbor	int *	Estimate of non-zeroes	0	10(9.9)
node_boundary	bool *	Bool vector for boundary	0	9(8.9)
f	double *	Right hand side Vector	0	3.5(3.5)
Other	-	-	0	9(8.9)
Total	-	-	2	101(100)

Table 4.1: Variables and their blame for run of FFP_SPARSE

variables in main. The "Base Data Centric" column represents explicit memory operations on the memory space of the defined variable. This means that the sample was taken when an assignment was occurring to an actual index within the array, and not to another statement from the backwards slice of that array that would have contributed to the blame. "Blame" refers to the number of samples in which blame was assigned to those variables.

One thing that stands out from this run is the lack of sample points (only two) where an explicit write was taking place to the arrays present at the top scope of *main*. This number includes any writes to these memory locations under all aliases as many of these arrays are passed as parameters throughout the program. In sparse matrix implementations, many of the computations take place behind layers of abstraction between the defined variable and where the work is actually taking place (i.e. the bookkeeping code that maintains the internal representation of the data structure). When blame mapping is introduced we get a clearer picture of what the program is trying to accomplish. The solution vector and the coefficient

matrix are the clear recipients for most of the blame of the program.

We manually inspected the code and the samples that were generated during the run. We mainly wanted to examine the interaction between the variables detailed in Table 4.1 and how their blame was determined. In the case for each of these variables (all declared within *main*), there was a series of calls within *main* that individually populated their values with minimal interaction between the variables themselves for the majority of the program. Finally, all of these variables became input parameters for the function that finally computes the solution for the system. At this point, much of the blame is transferred to the solution vector.

4.1.2 QUAD_MPI

QUAD_MPI [53] is a C++ program which uses MPI to approximate a multidimensional integral using a quadrature rule in parallel. While the previous program illustrated how a sparse data structure can be better profiled using variable blame, this program helps to illustrate how some MPI operations will be modeled. It is approximately $2,000$ lines of code and consists of 18 functions.

We ran the QUAD_MPI program on four Red Hat Linux nodes (using one core per node) using OpenMPI 1.2.8 and recorded a range of $94 - 108$ samples per node. All calls to MPI functions were handled by assigning blame to certain parameters based on the prototypes of the MPI programs utilized. The program exits after printing out the solution, represented by the variable *quad*.

The results for the run are shown in Table 4.2. The variables are listed in

Name	Type	MPI Call	Blame (per Node)				
			N1(%)	N2(%)	N3(%)	N4(%)	Total(%)
dim_num	int	Bcast	32.3	95.7	84.3	94.4	76.7
quad	double	Reduce	24.3	1.1	4.3	4.6	8.6
task_proc	int	Send	20.2	-	-	-	5.1
w	double*	-	14.1	-	-	-	3.5
point_num_proc	int	Recv	-	1.1	6.1	-	1.9
x_proc	double*	Recv	-	2.1	4.3	-	1.4
Other	-	-	3.0	-	-	-	0.7
Output	-	-	6.1	-	0.9	0.9	1.9
Total		-	100	100	100	100	100

Table 4.2: Variables and their blame for run of the QUAD_MPI

descending order based on the total amount of blame assigned across all nodes. For each variable, it is shown whether an MPI operation was a contributing factor, but not necessarily the only source, of the blame.

We manually examined the code to determine if our blame information was accurate and to try to understand why certain variables got assigned the blame they did. Most of the variable blame in this program is tied to the MPI operation they are a part of with little other computation occurring in the program. The variable with the most blame, *dim_num*, is due to program input from the master node at the beginning of the program, which causes the three other nodes to create an implicit barrier. The second highest blame count goes to *quad*, which is the variable that holds the output for the program so a high number is to be expected. This program has little data flow mappings contributing to blame, but is an interesting case study. Because certain variables are passed in as parameters to MPI operations (and blamed for them), we can use variables as an implicit aggregate for bunches of MPI operations operating on the same variable.

4.2 Uniqueness Factor

Traditional performance analysis tools primarily use code centric techniques. For data centric techniques to be useful, they need to produce insight that the code centric techniques cannot. In this section, we will discuss how we measure the "uniqueness" of information provided from data centric approaches.

For data centric approaches like those discussed in Section 2.1.1, every sample is attributed to a variable. This assignment is based on a memory related hardware counter triggering a sample due to that variable being accessed. This is information that is completely orthogonal to and can not be duplicated by code centric means.

There are cases where results from blame analysis may appear similar to values found from code centric techniques. This is because every sample is assigned at least to a local variable, and then bubbled up the call stack when applicable. This can lead to variables that are passed in as parameters having blame percentages very close to the percentages a code centric tool might give for the function that has that variable as a parameter (i.e. when a function operated on exactly one data structure). Figure 4.1 shows an example program that illustrates the relationship between functions (measured by code centric means) and variables (measured by variable blame) in terms of the percentages attributed to them.

The case where a variable might match up closely to the function is represented by $func1$ and the variable *blameSingle*. In this case, $func1$ has only one parameter, and *blameSingle* has only one function that it is passed into. Assuming that within $func1$ the data flow maps primarily to *blameSingle*, and not some local

```
void main()
{
   int * blameSingle;
   int *blame1, *blame2, *blame3, *blame4;
   int noBlame;

   func1(blameSingle);

   func2(blame1, blame2, blame3);
   func2(blame2, blame3, blame4);
   func2(blame4, blame3, blame1);

   func3(blame1, noBlame, noBlame);
}
```

Figure 4.1: Code displaying different uniqueness factors

variable, then the final percentages for time spent in $func1$ and the variable blame for *blameSingle* will mirror each other. We claim this information is "redundant" since the variable blame for $func1$ doesn't give us any additional information about the program that we couldn't have derived from code centric means.

The more common case is one where a function takes multiple parameters and the variables are passed in to multiple functions. This is shown by variables $blame1$, $blame2$, $blame3$, and $blame4$. They are all passed into $func2$ in different parameter combinations. In this situation, it would be very hard for a code centric tool to tease apart the amount of blame that is associated with each variable compared to the results that come back from the different $func2$ calls. Thus, we claim this information is "unique" to a data centric view.

We also include a call to $func3$ to show that we only consider "exit variables" to functions for this uniqueness classification. This means scalars that do not have a pointer, such as *noBlame*, are not a factor in determining uniqueness. Read only

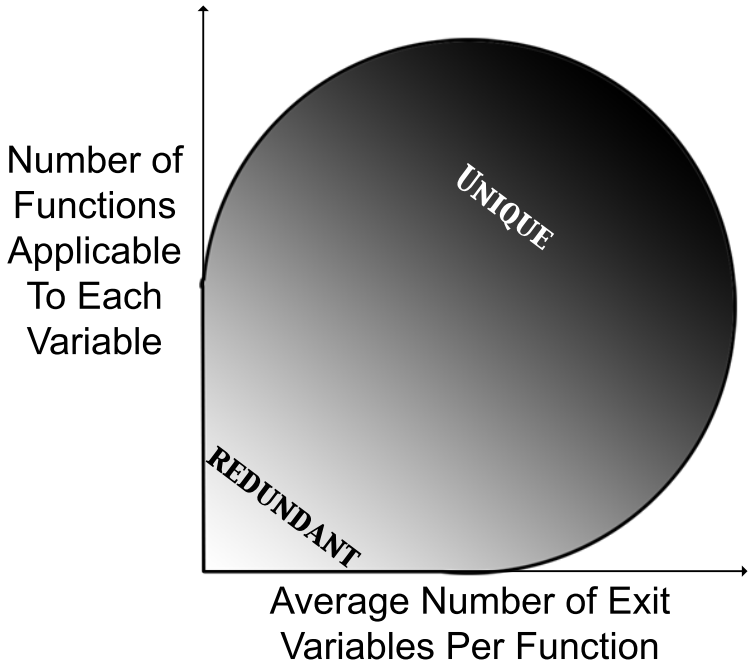


Figure 4.2: Uniqueness of Data from Variable Blame

parameters would also not be counted.

The relationship between code and data centric approaches is shown by the cartoon in Figure 4.2. We claim there is a sliding scale between redundant information and unique information. There are two factors that determine the "uniqueness" of information displayed in the figure. The first is the number of functions that take a variable in as a parameter. The second is the average number of exit variables per function. $func1$ and *blameSingle* are examples of redundant information. The situation detailed by $func2$ would be in the unique area of the graph. As illustrated in the figure, we have no hard cut-off point for where redundant data ends and unique data begins. It may be the case a function has only one exit variable but the majority of the blame goes to a local function. In that case, the blame attached to

the variable would be very different than the function that it is passed into. On the other side, a function may have multiple exit variables but have only one variable that receives the majority of the blame. In that case, the one variable would map closely to the function in terms of performance.

Redundant data does not necessarily mean useless data. There are cases where a variable may provide good insight into the program and be much more visible than when looking at the code centric view. One example of this would be a program with a series of vectors with various operations performed on them. If any given vector was only passed to one function, the data centric and code centric views would essentially be the same. At this point, the usefulness of the information is a function of how likely the user is to see the data based on the view. In a code centric view, this data may not ever be seen if it is buried deep in the call trace. In a data centric view, the measured vector may be a top level field for a highly utilized variable which would make it very visible. Our approach also allows variables to be filtered by type, meaning that looking at all of the vectors in the program based on this filter would also bring this data to the forefront.

When referring to the "uniqueness of a variable" for the rest of this dissertation, we will give two numbers. The first number is the count of functions that the variable is passed into uniquely within a call path. This means that in the above code example if $func1$ calls $func1a$ and passes $blameSingle$ as a parameter, that would not count since $func1$ is in that call path and had already been counted. The second number is the mean number of exit variables for the functions where the variable is passed in. We chose mean instead of median because one function

with a large number of blamed parameters may completely change how unique the blame data is. For compactness, the uniqueness factor will be given as the tuple $< Function\ Count,\ \ Mean\ Exit\ Variables >$ so the uniqueness factor for $blame1$ would be $< 3, 2.3 >$ For our running example, the uniqueness for the five variables is shown in Table 4.3.

Name	**Uniqueness Factor**
blameSingle	$< 1, 1.0 >$
blame1	$< 3, 2.3 >$
blame2	$< 2, 3.0 >$
blame3	$< 3, 3.0 >$
blame4	$< 2, 2.0 >$

Table 4.3: Uniqueness Values for Example Variables

We also distinguish cases where the variable is passed in as a parameter at a point in the call stack further down then where the variable is declared. This is a common case with complex types, where the parent variable is passed in to functions at the declaration site, and various fields may not be utilized until farther down the call stack. We add an asterisk (*) after the tuple to designate these cases. An example of this scenario is shown in Figure 4.3. The *se* variable is declared in the *main* function but for field *sx* it is not passed into a function until *case1*. The asterisk is used to distinguish that we are acknowledging the call to *foo* with the field as a parameter as separate to the call to *case1* with the container variable as a parameter.

There are also cases where a variable is never passed in as a parameter. This also can occur with complex types, where the parent is passed into a function and the

```
typedef struct {
int *sX;  int *sY;
} StructEx;

void case1(StructEx * se)
{
    foo(se->sx);
}

void case2(StructEx * se)
{
    se->sy[0] = 5;
}

void main()
{
    StructEx * se;
    ...
    case1(se);
    bar(se->sx);
}
```

Figure 4.3: Special Uniqueness Cases

field is written to without ever being passed on. We represent this with $< \infty, \infty >$. An example of this scenario is shown in Figure 4.3 in the *case2* function. The field *sy* is never passed as a parameter but is written.

Based on these special cases, the uniqueness factors for these variables would be the values shown in Table 4.4.

Name	**Uniqueness Factor**
se	$< 2, 1.0 >$
se→sx	$< 2, 1.0 >$ *
se→sy	$< \infty, \infty >$

Table 4.4: Uniqueness Values for Special Case Variables

As we previously stated, the uniqueness factor is a continuum, not a binary

```
int ** arrays, *arrSizes, i;
init(arrays, arrSizes);

for (i = 0; i < N; i++)
    qsort(arrays[i], arrSizes[i], sizeof(int), compare);
```

Figure 4.4: Sample program using qsort

threshold technique. However, a value of one or less for either of the components points to redundant information, whereas values of two or higher from both indicates unique information.

4.2.1 glibc Sort Case Study

The GNU C Library [61] contains an open source implementation for quicksort (contained in qsort.c). A small example utilizing this sorting function serves as a useful case study into how uniqueness might be a factor within a program and how variable blame compares to code centric methods in handling this use case.

The *qsort()* call takes four arguments. The first is a pointer to the data that is to be sorted. The second parameter is the number of elements to be sorted. The third parameter is the size of each element. Finally, the fourth parameter is the comparison function that determines how the sort is conducted. Our sample program using *qsort* is shown in Figure 4.4.

Our program contains an array of arrays which we loop through to sort each array individually. The uniqueness factor for the value of the *arrays* variable is $< 2, 1.5 >$. It is passed in as a parameter to *init* and *qsort* with those two functions having 2 and 1 parameters, respectively, that could be considered exit variables.

An important factor to notice is the *qsort* call occurs within a loop. By collecting runtime information about the loop index (the array offset) at each sample point, we can use the variable blame information to identify exactly how much time is spent sorting each array. Because of the call to *qsort*, exclusive code centric tools would not be able to provide that data. Inclusive code centric tools would also perform poorly with this scenario since each call has the same stack trace. Furthermore, since the call to *qsort* occurs at the same line every time, loop level and even individual line profiling would aggregate the data, losing the array specific data we seek.

To get the same information provided by variable blame from standard profiling techniques, we would have to perform caliper based instrumentation within the loop, surrounding the call to *qsort*. This would give us the proper values for this use case, since the operations on the arrays and the iterations through the loop match up in a 1:1 manner. In the majority of programs this mapping will not be the case. Multiple iterations of a loop may affect one index of an array (or data structure) while other elements may not be touched at all. While blame handles this style of problem natively, most code centric techniques would not be able to handle this type of problem.

We used four different arrays for our sample program. The first array was $50,000$ randomly generated elements. The next three arrays were $100,000$ elements. The second array was also randomly generated. The third array was already sorted. The fourth array was reverse sorted. We performed two runs. The first run we measured the time spent processing each array using our variable blame technique.

Array Description	Blame Time	Caliper Time
$50,000$ Random Elements	25.1%	25.3%
$100,000$ Random Elements	74.9%	74.7%
$100,000$ Sorted Elements	$< 0.1\%$	$< 0.1\%$
$100,000$ Reverse Sorted Elements	$< 0.1\%$	$< 0.1\%$

Table 4.5: Time spent to sort each array

The second run we measured the time spent using manually inserted caliper based instrumentation at the beginning and end of the loop that contained *qsort*. The results of the experiment are shown in Table 4.5.

The variable blame approach was able to achieve the same measurements as the caliper based instrumentation. It is important to note that the blame approach was able to perform the measurements as a part of its base operation. However, the results from the caliper based instrumentation was achieved from hand coded instrumentation that is not present in existing profiling tools. This simple example allowed us to compare the two approaches directly and allowed caliper based instrumentation to measure the same computation. However, it is not representative of the type of problem caliper based instrumentation can solve on a normal case. In real programs, where the uniqueness factor is higher and complex data types are involved, the ability to emulate the data being gathered by variable blame in a code centric manner becomes increasingly difficult and in most cases is not possible.

4.3 Case Studies

For all of the programs in this section, we ran on 32 Red Hat Linux nodes (one core per node) on the UMIACS bug cluster [15]. Each node contains two 2.6GHz

Xeon processors and 2GB RAM and are connected via Myrinet using OpenMPI 1.2.8. PAPI [11] is utilized to trigger samples based on the hardware counter measuring total cycles reaching a threshold, $1,073,807,359$, which is a large prime.

For each program, we have chosen the variables (and their fields) that have the highest degree of importance (highest blame percentage) for the program. The variables are listed in Table 4.6 for HPL, Table 4.7 for SMG2000, and Tables 4.8 and 4.9 for PFLOTRAN. The name column represents the name of the variable. In cases where the variable is a field tied to another variable, a $\hookrightarrow$ symbol is used to represent that relationship (with the parent variable listed above it in the table). The type is the defined type as given by our analysis. The Blame % is the mean of all assigned blame across the cores (32 for these results). None of the variables had significant variance between cores. The "Where Defined" column refers to the point in the code where the variable was declared, and not necessarily where the variable is first used.

4.3.1 HPL

HPL is an implementation of the "High Performance Computing Linpack Benchmark" that solves a linear system in double precision on distributed systems [30]. HPL offers a variety of attractive features as a test program for blame mapping. It utilizes MPI and BLAS calls and has wrappers for the majority of the functions from both libraries. HPL is interesting to examine because it is similar to many parallel frameworks in that MPI communication is completely hidden from

the user. This means tracing MPI bottlenecks using traditional profiling techniques may technically give you information about where the bottleneck is occurring. However, that information may be useless because the MPI operations are buried deeply enough in complex data structures that knowledge of how these bottlenecks affect variables at the top levels of the program is difficult to discover.

Name	Type	Blame % (Mean)	Where Defined	Uniqueness
mat	HPL_S_pmat	95.6%	main→HPL_pdtest	$< 18, 3.3 >$
panel	HPL_T_panel**	68.1%	HPL_pdgesv0[1]	$< 2, 1.0 >$
panel[0]	HPL_T_panel*	68.1%	HPL_pdgesv0[1]	$< 9, 1.8 >$
↪A	double*	35.0%	HPL_pdgesv0[1]	$< \infty, \infty >$
↪U	double*	27.5%	HPL_pdgesv0[1]	$< \infty, \infty >$
↪L2	double*	25.4%	HPL_pdgesv0[1]	$< 3, 2.0 >^*$
grid	HPL_T_grid	5.34%	main	$< 4, 3.67 >$

Table 4.6: Variables and their blame for run of HPL

The *main* function serves primarily to read in program specifications and iterate through the tests, which have their own output. For this reason, many of the more interesting variables are defined deeper down the call stack. The blame points for this program are *main*, *main*→*HPL_pdtest*, and *main*→*HPL_pdtest*→*HPL_pdgesv*→*HPL_pdgesv*0.

mat The variable *mat* is the focus of all of the computation in the program and therefore receives the majority of the blame. *mat* itself is a container for the raw data for the matrix A, solution vector X, and right hand side b, plus all of the metadata. The matrix information is regenerated randomly at each time step. The uniqueness factor shows *mat* to be unique, being passed

[1]Full path is main→HPL_pdtest→HPL_pdgesv→HPL_pdgesv0

into 18 functions. However, with *mat* taking up such a large part of the computation it becomes obvious that *mat* is tied very closely to the function *HPL_pdtest*, which is the workhorse function for the program. This kind of special redundancy can sometimes occur for variables defined very close to *main* in the call trace when a disproportionate focus is placed on a small subset of variables.

panel The panel itself (after accessing the first index in the panel array) has fields associated with it that can provide insight into the program. The panel components have a high uniqueness factor and the fields that are a part of each panel give insight into the functionality of the program. The program uses LU factorization, represented in each panel by the A, U, and $L2$ variables. The U variable is a pointer to an offset within A, and much of the blame overlaps between the two. All of the variables within panel are pointers to offsets within the top level *mat* variable.

grid The variable *grid* is the container variable for all of the functionality involved in setting up the computational grid (mostly low level MPI operations) for the parallel solver to operate. It is passed into most of the functions in the program, mostly in a read only context with a write to *grid* happening occasionally. Most of the blame associated to the variable comes from the occasional calls to *MPI_Barrier*.

The profiling numbers for the *panel* variable is the most telling data about what is going on in the program. The blame data for this variable serves as an

implicit aggregate to many of the MPI operations within the program, and thus looking at the data at a per core level could yield information about possible load imbalances and other inefficiencies. Conveniently, many underlying parameters involved with *panel* are all customizable. The configuration file loaded for each HPL run has various panel parameters specified, ranging from individual panel size to the choosing which algorithm to use to broadcast the panels. The modification of any of these parameters will change the blame percentages associated with the *panel* variable.

4.3.2 SMG2000

SMG2000 [10,59] is a benchmark that uses a parallel semicoarsening multigrid solver for those linear systems that arise from finite difference, finite volume, or finite element discretizations of the diffusion equation on logically rectangular grids. It is written in C and performs data decomposition by dividing the grid into logical chunks of equal size. It is a good candidate to profile using variable blame due to its hierarchical data structures.

solver The *solver* variable is the top of the data hierarchy for the program. It has a significant number of void* fields that map to different data types depending on some of the input parameters to the program. The variable *solver* itself starts as a void* and is cast based on the solver type chosen at runtime.

relax_data Relaxation operations are a core mechanism for the solver in this benchmark. There are multiple functions that deal with relaxation. The

blame for this variable is useful because it serves as an implicit aggregate for all of the relaxation operations that take place in the entire program.

solve_data Each variable of type *hypre_SMGData* (such as the top level solver) has a field of type *hypre_SMGRelaxData* which itself has a field of type *hypre_SMGData*. Using the blame mappings, one can determine at what point in the relax-solve cycle the data is actually being written. The solve_data variable is passed into fifteen unique functions with an average of 1.9 writable parameters. By our metric, the uniqueness of this variable is very high meaning that it would be hard to calculate this same information with code centric means. Furthermore, by being a container variable for various work vectors, discussed below, it provides a useful aggregate for the kinds of operations found in the relax-solve cycle.

Name	Type	Blame % (Mean)	Where Defined	Uniqueness
solver	hypre_SMGData*	98.6%	main	$< 4, 2.8 >$
↪relax_data_l[0]	hypre_SMGRelaxData*	84.0%	main	$< 22, 1.7 >*$
↪solve_data[*]	hypre_SMGData*	80.0%	main	$< 15, 1.9 >*$
↪x_l	hypre_StructVector**	45.6%	main	$< 11, 4.0 >*$
↪r_l	hypre_StructVector**	30.8%	main	$< 4, 5.0 >*$
↪b_l	hypre_StructVector**	4.7%	main	$< 9, 4.2 >*$
↪A_l	hypre_StructMatrix**	2.4%	main	$< 7, 4.3 >*$

Table 4.7: Variables and their blame for run of SMG2000

Most of the processing takes place to populate various fields for *solve_data*. The benchmark is memory access bound, with various work vectors getting multiple writes. A direct data approach would be able to identify some of these writes, but would not be able to perform the additional mappings that our approach allows.

The importance of this additional mapping capability comes into play when dealing with the *solve_data* vectors. The benchmark switches between relaxation and solving depending on the data decomposition. The blame attributed to vectors map closely to certain operations within the program. The writes to the x_l vector correspond to the computation to perform the cyclic reductions. The blame attributed to the r_l vector is an aggregate all of the writes that take place when calculating the residual.

4.3.3 PFLOTRAN

PFLOTRAN [52] is a large-scale parallel 3-D reservoir simulator that can model multiphase reactive flows in geologic formations based on continuum scale mass and energy conservation equations. PFLOTRAN itself is written primarily in FORTRAN, with a few auxiliary functions and wrapper functions to integrate external libraries in C. It employs PETSc's solver framework, written in C, and also utilizes MPI, BLAS, and LaPACK. This is another program that is appropriate for our technique because of the links between mathematical constructs and variable types, mainly those involving PETSc. This program also hides much of the parallel computation inside calls to PETSc operations. For this reason it is very desirable to map performance information to these variables.

PFLOTRAN is hierarchical in terms of its data types, essentially written as an object-oriented program. The *simulation* variable, defined in *main*, has virtually all of the blame in the entire program assigned to it. The *simulation* variable consists of three main fields, $flow\_stepper$, $tran\_stepper$, and $realization$. The first two are

the time stepper container types and contain, as part of their hierarchy, most of the PETSc variables. The final field is a holder for most of the useful information that is calculated within the program.

The internal PETSc variables are utilized by the solver within the time stepper. The solver performs a residual function evaluation and performs the calculation of the Jacobian matrix. This information than feeds back into the higher level PFLOTRAN variables.

The top level PFLOTRAN variables serve mostly as containers for information and can give useful insight into the program.

Name	Type	Blame % (Mean)	Where Defined	Uniqueness
simulation	simulation*	99.3%	MAIN__	$< 3, 1.3 >$
↪flow_stepper	stepper*	87.9%	MAIN__	$< 1, 3.0 >$
↪solver	solver*	87.3%	MAIN__	$< 3, 1.0 >^*$
↪snes	_p_SNES*	87.3%	MAIN__	$< 2, 3.5 >^*$
↪realization	realization*	7.0%	MAIN__	$< 1, 3.0 >$
↪discretization	discretization*	3.1%	MAIN__	$< 25, 4.8 >^*$
↪field	field*	2.0%	MAIN__	$< \infty, \infty >$
↪tran_stepper	stepper*	4.4%	MAIN__	$< 1, 3.0 >$
↪solver	stepper*	4.0%	MAIN__	$< 3, 1.0 >^*$
↪snes	_p_SNES*	4.0%	MAIN__	$< 2, 3.5 >^*$

Table 4.8: Variables and their blame for run of PFLOTRAN

simulation We have already discussed how the *simulation* variable (of simulation type) serves as the root variable for the entire function.

flow_stepper and tran_stepper The two stepper variables maintain all of the data calculated during the various time steps. PFLOTRAN has computation for multicomponent reactive flow ($flow\_stepper$) and transport ($tran\_stepper$)

at each time step. Since the time step data is written to both of the stepper variables, these variables give the percentage of time spent populating the data structures that store the values for the respective multicomponent flow and transport operations.

realization The *realization* variable is the container variable for the discretization and field variables associated with the simulation. The *realization* object contains auxiliary data needed by the solver, but the writes to this object are limited. For this reason, the potential for optimizing this variable is limited.

The PETSc variables are internal to the stepper variable hierarchy, which each contain a PETSc non-linear solver context (SNES) as one of their fields. Table 4.9 is a continuation to the hierarchy of the $flow\_stepper$ variable, specifically, $simulation \rightarrow flow\_stepper \rightarrow solver \rightarrow snes$.

PETSc variables, in general, are very customizable. By changing certain factors during variable initialization, the underlying computation can change from serial to parallel and the data layout can be significantly altered. Furthermore, other parameters can completely change the underlying algorithm utilized to compute the contents of the variable. For example, using the PETSc options database, a user could change the type of preconditioner used at runtime. This would result in changing which functions are utilized by the preconditioner, as well as the "data" variable within the "pc" object being cast to a different data structure. Therefore, depending on the parameters passed to the initialization function for these PETSc variables, the blame percentage could change significantly.

Name	Type	Blame % (Mean)	Where Defined	Uniqueness
snes	_p_SNES*	87.3%	MAIN__	$< 2, 3.5 > *$
↪ksp	_p_KSP*	83.3%	MAIN__	$< 3, 3.3 > *$
↪work	_p_Vec**	71.3%	MAIN__	$< \infty, \infty >$
↪work[0](R)	_p_Vec*	32.6%	MAIN__[2]	$< 6, 3.3 >*$
↪work[4](S)	_p_Vec*	19.2%	MAIN__[2]	$< 4, 3.7 >*$
↪work[2](V)	_p_Vec*	18.8%	MAIN__[2]	$< 5, 3.4 >*$
↪work[5](P)	_p_Vec*	1.8%	MAIN__[2]	$< 4, 3.8 >*$
↪work[3](T)	_p_Vec*	1.7%	MAIN__[2]	$< 4, 4.3 >*$
↪work[1](RP)	_p_Vec*	0.1%	MAIN__[2]	$< 2, 3.0 >*$
↪pc	_p_PC*	48.7%	MAIN__	$< 20, 2.9 >*$
↪mat	_p_Mat*	16.7%	MAIN__	$< 5, 3.0 >*$

Table 4.9: PETSc Variables within a run of PFLOTRAN (drill down from the solver for flow_stepper from Table 4.8)

snes The *snes* variable is the container variable for a PETSc non-linear solver. Modifying parameters during the creation of this variable will impact the solver context, thus significantly altering the blame percentage associated with the *snes* variable. This variable falls in the redundant camp, however, as the computation used to populate the *snes* variable almost directly mirrors the function $SNESSolve$.

ksp The *ksp* variable is the container variable for computation involved in solving a linear system. *ksp*, and its internal components, are very customizable and can yield very telling information about what is going in the program. Unfortunately, like *snes*, the blame percentage is redundant and maps to the $KSPSolve$ function. However, the internal components to *ksp* provide unique data and are also tunable.

[2]The work field is a descendant of the *simulation* variable, declared in MAIN__. The local vectors that are aliased to the work array are declared in MAIN__→stepperrun→stepperstepflowdt→snessolve_→SNESSolve→SNESSolve_LS →SNES_KSPSolve→KSPSolve→KSPSolve_BCGS

work The *work* variable is actually a pointer to a series of vectors. The number of vectors and how they are utilized is completely dependent on how the user configures their *ksp* object. In the case of PFLOTRAN, these vectors are generated in a worker function and aliased to different indexes of the work array. This is a good showcase of the successful alias analysis of our system, as the local vector name (in parentheses in the table) and the offset of the work array is correctly determined through static analysis. As the work is performed by PETSc vectors, blame percentages for the work may be affected by the customization of these vectors. Among the options upon initialization is whether the user wants the array to be distributed or not.

pc The preconditioner for the Krylov space methods (solving approach used by KSP) can be chosen among many provided by PETSc. The choice of preconditioner can make a difference in the blame associated with the *pc* object, and also the *ksp* object. The preconditioning of a matrix is done with the idea that the time spent doing the preconditioning will be made up with a faster solution to the linear system. The *pc* variable has a very high uniqueness factor as it is passed in as a parameter to 20 unique functions and each function has an average of approximately 3 writable parameters. The operations on the preconditioner are an integral element of the computation, making it an excellent variable to focus on for tuning.

mat The *mat* object is a core data structure within the *pc* object. The *pc* also uses some of the generic *ksp* work vectors in its computation. Matrices in

PETSc are customizable. One of the interesting ways to manipulate matrices in PETSc at initialization time is to choose whether to make them dense or sparse (with choices on what kind of sparse matrix to use). Obviously, the choice of how to represent the matrix internally can affect the blame associated with that variable. Although this is the only PETSc matrix we are explicitly discussing here, the Matrix objects (along with Vectors) are very common in PETSc computations.

Much of the blame for PFLOTRAN is assigned to the underlying PETSc objects, which execute non-linear solvers at each time step. PFLOTRAN utilizes custom implementations for its preconditioners and solvers. Furthermore, the underlying data structures for the PETSc vectors and matrices are optimized specifically for the data sets used by PFLOTRAN. Our analysis could prove useful for PFLOTRAN developers when trying out new custom configurations as our analysis could localize the time spent populating individual data structures. This could be compared across runs among different candidate customizations. We can also use our analysis to optimize performance on different hardware configurations based on information our analysis gives us. For the rest of this section, we will use variable blame to narrow the parameter space and use that information to improve performance.

The following experiments were first performed on either 4 or 16 bug cluster nodes where we used one core per node with the default configuration file that was packaged with the input data. Two input sets (100x10x10, 100x100x100) were

sample data sets derived for testing purposes. The final input set (30x30x15) was actual data from the Hanford simulation.

For these experiments, we did not modify any of the parameters for SNES or KSP values. Performance gains can be achieved by modifying the options for these variables (such as max iterations for SNES or solver type for KSP), but these modifications could affect the accuracy of the solver. For consistency, we only modified the behavior of variables that would have no bearing on the accuracy.

After evaluating the performance on the bug cluster, we also performed the experiments on up to 512 cores on Carver [17], a NERSC machine. Carver has 400 compute nodes with 2 quad-core Intel Xeon 5500 ("Nehalem") 2.67 GHz processors for $3,200$ total cores. The memory for each node ranges from $24 - 32$ GBs.

4.3.3.1 Example 100x10x10

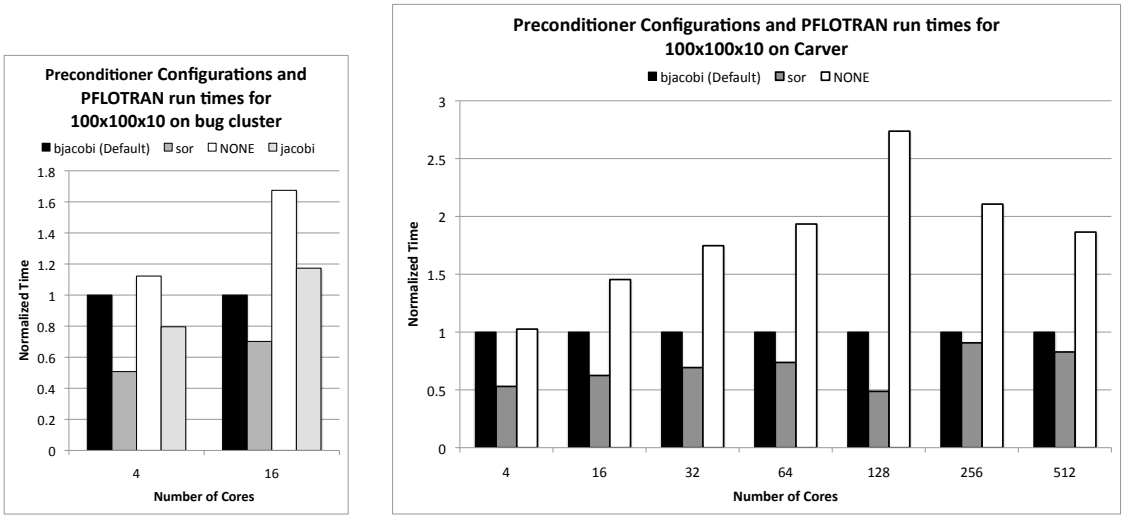
For this data set, we approached the problem by looking at the flat view of all the blamed variables within the program. These values are show in Table 4.10.

Name	Type	Blame % (Mean)	Where Defined
simulation	simulation*	99.3%	MAIN__
↪flow_stepper	stepper*	84.2%	MAIN__
↪solver	solver*	84.2%	MAIN__
↪snes	_p_SNES*	84.2%	MAIN__
↪ksp	_p_KSP*	63.4%	MAIN__
↪pc	_p_PC*	42.1%	MAIN__

Table 4.10: Variables and their blame for 100x10x10 data set run on four cores

We have already stated we are not going to modify the behavior of the *snes* or *ksp* variables so the next variable we could possibly optimize is the *pc* variable. For

this data set, the *pc* variable takes 42.1% of the time. Since the preconditioner serves as a means to make the solver complete faster (and subsequently the whole program) it is counterproductive to have a preconditioner consume a large percentage of the computation time unless it can make up the difference in how much it speeds up the solver.



a) b)

Figure 4.5: 100x100x10 performance on a) bug cluster and b) Carver

One option is to eliminate the preconditioner. The other option is to find a preconditioner that runs faster but still manages to speed up the solver. These two alternatives are shown in Figure 4.5. The runs with no preconditioner run slower for both the 4 and 16 core cases. For 4 cores, the *jacobi* and *sor* preconditioner improve overall performance. For 16 cores, the *sor* preconditioner performs better.

When performing the same runs on Carver, similar trends occur. Using no preconditioner performs worse on most cases, while the *sor* preconditioner performs better through 512 cores. The absolute times are presented in Figure 4.6. We can

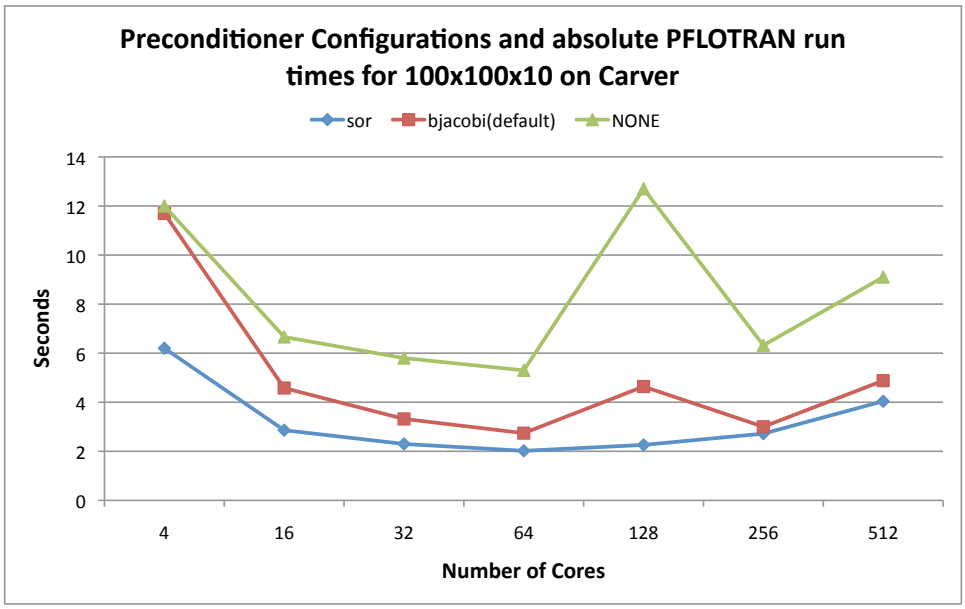


Figure 4.6: Absolute times for 100x100x10 runs on Carver

see this data input set has poor strong scaling performance when extending out to 512 cores. We should note the scaling performance is not the focal point of these experiments. For these experiments, we are mostly concerned with displaying the pair-wise comparison between different preconditioner inputs for the the different core counts.

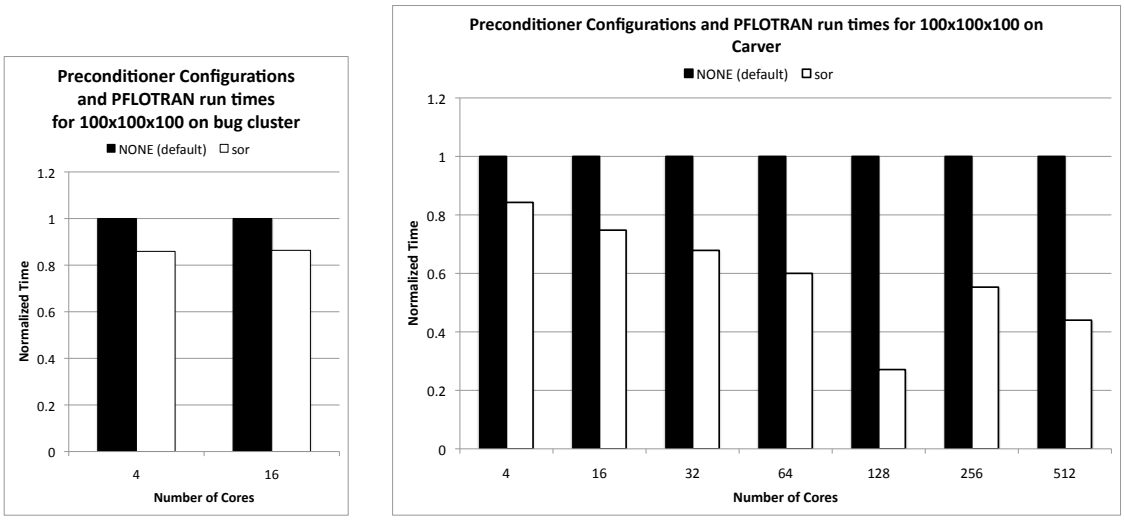
4.3.3.2 Example 100x100x100

For this data set, we utilized the type filter. The counts for the variables that have PETSc types are in Table 4.11.

Type	Count
_p_PC*	2
_p_SNES*	5
_p_Mat*	20
_p_Vec*	131

Table 4.11: The number of variables with each associated PETSc types

There are only two variables of a "_p_PC" type. When looking at the blame values for these two preconditioners we see that the values are 0.43% and 0.45% for the *flow_stepper* and *tran_stepper* preconditioner, respectively. Therefore, for this example, we have the opposite problem we had from the last problem, mainly that the preconditioner is taking much less time than we would expect. When looking deeper, we find that the preconditioner had been been disabled for both solvers in one of the input configuration files. The type view is helpful in this case because in a flat view these variables would have been buried down the list.



a) b)

Figure 4.7: 100x100x100 performance on a) bug cluster and b) Carver

Figure 4.7 shows the normalized overall runtime when using a preconditioner(*sor*) and using the default configuration of no preconditioner on the bug cluster. For that configuration, using the *sor* preconditioner makes the entire program run faster than not using any preconditioner at all. When performing the same runs on Carver, the difference is even more apparent. The absolute times for Carver are presented in

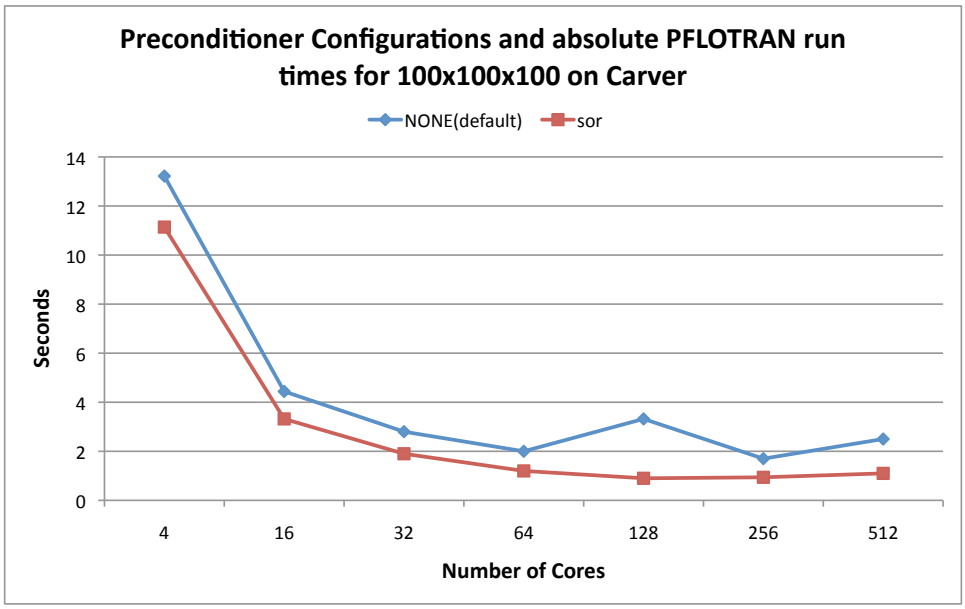
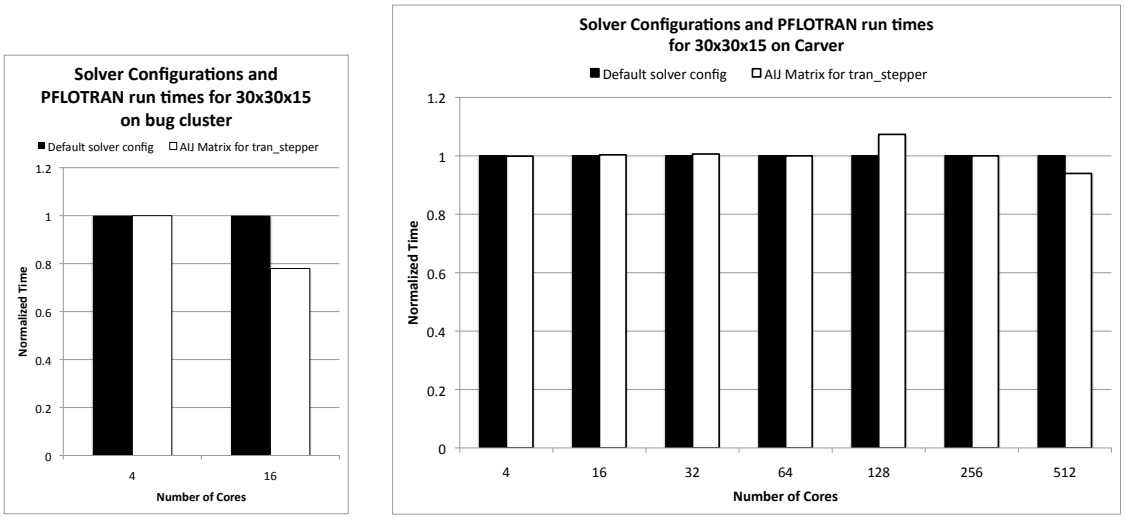


Figure 4.8: Absolute times for 100x100x100 runs on Carver

Figure 4.8.

4.3.3.3 Hanford 30x30x15

For this example, we will use the comparison view. Using two runs, with 4 and 16 cores on the bug cluster, we will compare how the computation to populate a variable (as given by blame) scales across different variables. When looking at absolute time (taking the percentage blame times the per core runtime), there are 3 variables where the time to populate a variable increases (per core) as the number of processing cores increases. These 3 variables are $flow\_stepper$, $flow\_stepper \rightarrow solver$ and $flow\_stepper \rightarrow solver \rightarrow snes$. We look at the input configuration files to see what makes this solver behave differently than the $tran\_solver$, which is parallelizing well. The input configuration is explicitly setting the $flow\_stepper$ to the aij matrix format, where the default in the program is the $baij$ format. Switching this over to



a) b)

Figure 4.9: 30x30x15 performance on a) bug cluster and b) Carver

the default *baij* for the *flow_stepper* solver does make the matrix parallelize better (the 16 core case takes less time per core to calculate the solver than the 4 core case), but switching the matrix configuration makes the overall program run slower for both hardware configurations. Using the knowledge that the default overriding input configuration for the flow_solver increases overall program performance, we tested to see if using that configuration for the *tran_stepper* solver would increase performance as well. As shown in Figure 4.9, the 4 core performance for the bug cluster remains the same with an improvement in the 16 core performance.

The comparison view for carver shows different results than the bug cluster, with zero variables increasing their blame values across runs from 4 to 16 (the three *flow_stepper* variables' computation time decreases per core on carver). The experiments with the modified configuration reflect this difference as well. The runs are fairly comparable regardless of configuration through 512 cores. Because the

plots are virtually identical, we do not present the absolute numbers for Carver for these experiments.

4.4 Blame Overhead

The overhead for performing blame analysis can be divided into three main areas. The first is the time it takes to do the intraprocedural analysis and storing the results of that analysis, before the target program is executed. The second is the overhead to do the stack walk at each of the samples during runtime. The final measure of overhead is the post processing step, which can be done in parallel immediately after the run completes. This step also raises concern of scalability for both runtime and the sizes of files that are generated while profiling HPC applications.

4.4.1 Pre-Run Static Analysis

We have a one time cost for running the initial intraprocedural analysis, which forms the basis for our blame mappings. Once this initial analysis is run, we only need to run it again on a module if a change is made to the source within that module. Table 4.12 displays the time required to run the initial blame analysis for eight shared libraries utilized by PETSc. The runs were performed on one Carver core.

The table shows the sequential time taken to perform the static analysis. This work could be parallelized at the library or module level by simply assigning one core per input file. Parallelism could also be exploited at the function level as well. This

Library Name	Total Time (seconds)	Num Functions	Avg Time (functions)	Max Time (functions)
mat	155	1850	0.08	4
ksp	60	1223	0.05	1
sys	58	1235	0.05	2
dm	40	489	0.08	14
vec	36	690	0.05	1
ts	18	268	0.07	1
snes	12	315	0.04	1
contrib	2	35	0.07	1
Total	381	6105	0.06	14

Table 4.12: Blame Static Analysis Run Times for PETSc libraries

would only involve doing a split of a module into separate input files by function.

The average time to process a function is 0.06 seconds. The max time is 14 seconds, which occurs with the *DACreate3d* function within the *dm* library. This function is over $1,500$ lines of code and has multiple nested loops and conditionals, which causes a bottleneck with the implicit processing.

4.4.2 Runtime

The runtime overhead is a product of the sampling rate and the time it takes to walk the stack at each sample point. All other computation is pushed to pre and post run. When using a time-based metric, we try to have between $100 - 200$ samples per second. This leads to an overhead of approximately $30 - 35$ percent. This overhead is almost entirely caused by the time needed to walk the stack, as the stack walk overhead alone is consistently within one percent of the overhead of our full implementation. Therefore, reducing the runtime overhead becomes an issue of optimizing the stack walk. Froyd et al [23] showed that by representing the samples

as a calling context tree, as opposed to explicitly storing the call stack for each sample, overhead can be reduced to approximately $2-10$ percent. Tallent et al [60] perform a linear scan of the object code, apply heuristics to reconstruct the stack, and then cache the results. They average approximately a 2 percent overhead with their approach. Future work would involve applying either of these optimizations to our current stack walking techniques.

4.4.3 Post-Mortem Analysis and Scalability

We examined how the blame post-processing scaled in terms of both time and aggregate file size. The figures presented in this section are presented from data taken from the Carver runs presented in Section 4.3.3, specifically from the analysis of the 100x100x10 data set with the default *bjacobi* preconditioner.

4.4.3.1 Processing Time as Core Count Increases

Our post processing falls into three steps. The first step is the per-thread resolution of samples to variables relying mainly on the blame analysis we complete pre-run. At this stage, the drill down information for each sample and stack frame is still intact. The second step is a mapping step that takes the resolved blame file from each thread and outputs the percentage blame for each variable over the course of the entire program. Finally, we have a reduction phase that takes the mapped output and creates one file that represents the aggregate blame for each variable across all threads. Steps 1 and 2 are done in parallel, while step 3 is done serially

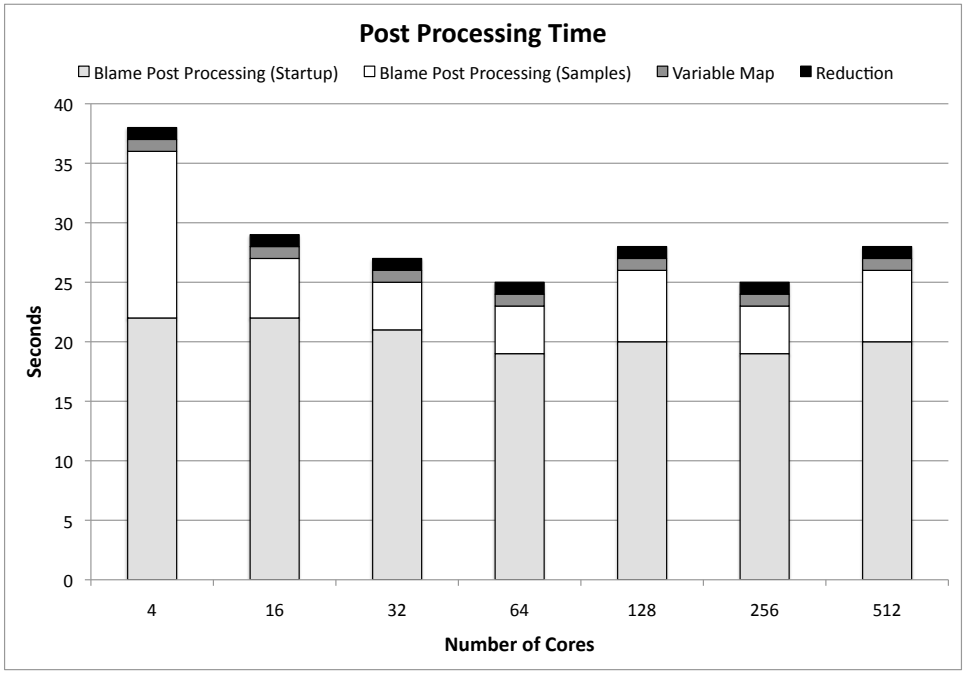


Figure 4.10: Post Processing Time for Each Stage

on one core. We detail the time it takes to perform these steps on one of our runs in Figure 4.10.

The main blame resolution phase, which is done in parallel, scales well with the increase of cores. For the purpose of Figure 4.10, we break this phase into two distinct steps. The first step is the startup phase, in which the program loads in the results of the pre-run static analysis. This step takes approximately the same amount of processing time, regardless of the number of samples. The second step is to actually resolve the blame for each sample. This is an embarrassingly parallel step due to the fact that the raw context sensitive samples are independent from each other for each thread. Because this is an embarrassingly parallel step, the processing time is tied more closely to the average number of samples being processed on each core, rather than any increase in core count. The mapping phase consistently takes

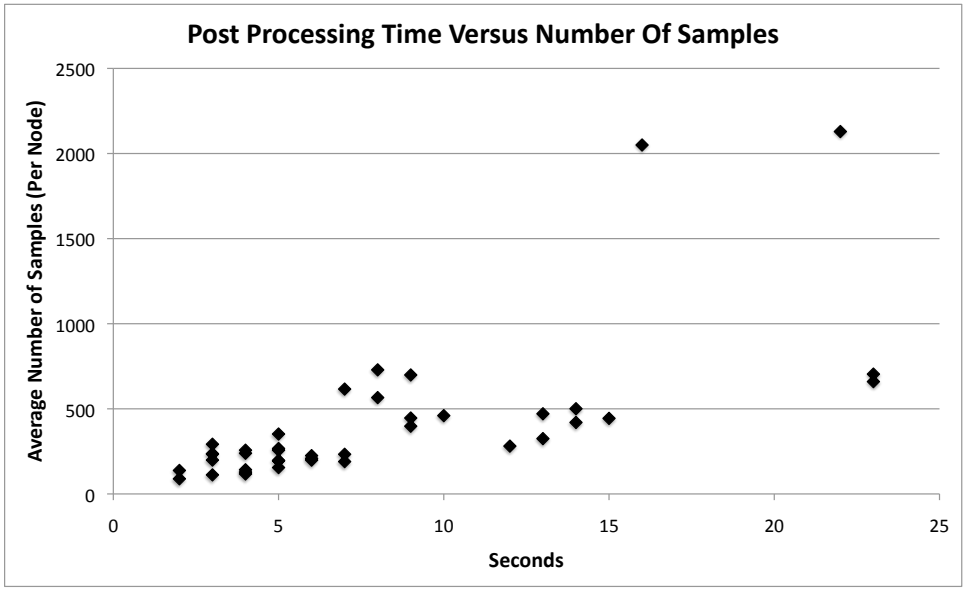


Figure 4.11: Post Processing Time versus Number of Samples Per Core

under a second for all core counts. This phase is also done in parallel. The reduction phase is not done in parallel and also consistently takes less than a second.

4.4.3.2 Processing Time as Samples Increases

Figure 4.11 shows the processing time for the blame resolution phase as the number of samples increases. This graph contains data from each of the core count and input set combinations for the PFLOTRAN runs in Section 4.3.3. The relationship between processing time and number of samples is not strictly linear. The main factors that create the outliers come from the average size of the stack trace and the number of side effects that need to be dealt with at each frame in the stack trace.

4.4.3.3 Aggregate File Sizes Across All Cores

A factor with scalability is the aggregate size of the output files as the number of cores increase. This is displayed in Table 4.13. The different types of output files are:

- **Raw Stack Traces** The raw stack frames (PC address) for each sample.
- **Resolved Blame** The list of blamed variables for each frame in the stack.
- **Blame by Variable** For each core, a list of local variables and its percentage blame for that core.
- **Aggregate Blame** For the entire system, the percentage blame for each local variable.

For all of our files, the files are output as human readable text. The file sizes could be reduced at each stage by using binary output. The stack trace file sizes could be curbed slightly by combining common frames between successive samples. Once the blame is resolved, these files can be safely deleted. The resolved blame files are the largest, but are also optional. For those runs where drill down data at the stack frame level is not required, the generation of these files can be completely bypassed. The aggregate blame files are what is fed into the GUI and only take up 0.02 MBs total across all nodes. This file is the result of the Map-Reduce operation where the file represents the blame for the variables across all cores.

	Aggregate Files Sizes (MBs)			
Num Cores	**Raw Stack Traces**	**Resolved Blame**	**Blame by Variable**	**Aggregate Blame**
4	1.9	25.6	0.9	0.02
16	2.3	29.1	2.2	0.02
32	4.5	45.1	4.5	0.02
64	8.6	73.5	8.3	0.02
128	32	166	17	0.02
256	93	523	33	0.02
512	193	1800	65	0.02

Table 4.13: File Sizes at Different Stages of Post Processing

4.5 Summary

In this chapter, we have presented sets of experiments detailing the information blame can provide and the overhead of running blame. We have established a metric for describing the 'uniqueness' of the information provided by running blame analysis on a program in comparison to the information given by a standard code centric tool. This metric helps illustrate that redundant data is not being calculated. The uniqueness factor was one element discussed in a series of case studies, where we used variable blame to better program understanding. We followed up these case studies by using blame analysis to improve performance on the PFLOTRAN program across different hardware configurations. Finally, we examined the overhead of using variable blame at the preprocessing, runtime, and post-processing stages.

Chapter 5

Approximate Data Centric

Variable blame is mainly concerned with mapping time based metrics to data centric objects. Most of the prior work for data centric mappings concerned mapping data centric metrics (such as cache misses) to variables. As outlined in Section 2.1.1, these approaches have certain limitations, mainly concerning the need for specific hardware support. Using much of the same analysis approaches as we used for variable blame, we have created an alternative approach for these data centric mappings.

Our technique is concerned with two main ideas. First, we want to have a generic approach that works on any architecture, regardless of hardware support for negating skid. Our only hardware requirement is that some data centric hardware counter exists on the system for measuring cache or TLB misses. Second, we want to minimize program perturbation and program instrumentation as much as possible. We do this by using sampling instead of direct measurement. Furthermore, our instrumentation is at the binary level and is limited to inserting two calls at the beginning and ending of main. These calls simply assign which hardware counter and threshold will be used to trigger the samples.

In this chapter, we will use the term "direct data centric" to refer to the approach utilized by Cache Scope and HPCToolkit for their data centric profiling.

This is because the performance counter relating to each variable is only incremented during a direct access of that variable. This is verified by the effective address information provided by the hardware. Our approach will be designated as "approximate data centric" since we are using software techniques to approximate which variable triggered the cache miss.

5.1 Intraprocedural Static Analysis and Execution

The majority of the analysis occurs at this step before the program is run. Using the LLVM intermediate format [39], we can analyze different properties at the program, function, and variable level. For the variables, we record the line number information for each read and write performed to that variable. At the function level, we can record the line numbers within every loop within the function. Loop information is utilized in tuning the results and is discussed later in this section. We also use the LLVM information to perform intraprocedural alias analysis. The alias analysis is a very important factor in generating approximate data centric values. Consider the snippet shown in Figure 5.1.

```
1  int * x = (int *) malloc (sizeof(int) * N);
2  int z = x[N-1];
3  int * y = x;
4  z = y[N-2];
```

Figure 5.1: Code snippet highlighting aliasing

In a direct data approach, a miss on line 2 or 4 would be attributed to variable x, allocated on line 1, because effective addresses of the read would resolve to the

memory range allocated to x. In an approximate approach without alias analysis, a miss on line 2 would be attributed to x and a miss on line 4 would be attributed to y. With alias analysis, x and y would be identified as aliases to one another, and the results would line up with the direct method. Interprocedural alias analysis is also performed after we gather the runtime information and know the full call trace for the samples.

The execution stage is performed in the same way as it is for variable blame. We use instrumentation to enable event driven sampling and we record these samples at runtime.

5.2 Approximation Techniques and Post-Processing

Our post-processing step takes the raw context sensitive samples and the stored intraprocedural analysis to determine the approximate data centric values for each of the variables per node. We start by generating a raw approximate assignment to the variables. We then apply further passes to refine those assignments.

5.2.1 Raw Approximate Assignment

After resolving the addresses to functions and line numbers, the raw values attributed to each variable are calculated by taking the number of misses per source line and dividing them equally among the reads for a line. Further passes may modify these values, but the starting point for our approximation uses this method. Only unique reads are counted towards the total. Given the snippet in Figure 5.2, assume

	Approx. Misses Raw			
Line	x[]	y[]	a	b
2	0.25	0.25	0.25	0.25
3	0	0.5	0.25	0.25
4	0.5	0	0	0.5

Table 5.1: Cache misses assigned using 'Approximate Raw' technique

```
1  for (a = 0; a < b; a++) {
2         i = x[a] + y[b]
3         j = y[a] + y[b]
4         k = x[b] + x[b]
5 }
```

Figure 5.2: Sample Code Snippet

exactly one cache miss occurs at each line in the body of the loop making three total misses. Our approach would initially attribute the following miss numbers, shown in Table 5.1, to the variables within the above code snippet.

Line 2 has the misses appropriated evenly across the two arrays and two integers that index the arrays. Line 3 has half of the misses appropriated to array y, because y has two reads indexed by two separate integers. In the case that y was indexed twice by the same integer, y would only have one read counted. For the accesses on line 3, it is possible that a equals b, meaning the same memory location would be accessed twice. However, we assume each access to be different unless a constant is used to access the array. Finally, for line 4 the allocation is split between x and b. Four reads take place on this line, but only two unique memory addresses are read.

```
1  for (i = 0; i < N; i++)
2   {
3       int x = a[i];
4       x += b[i];
5       x += c[i];
6   }
```

Figure 5.3: Simple Loop Code Snippet

5.2.2 Loop Compensation

The values assigned to the variables are initially divided equally based on the reads per source line. However, there are common cases with scalar variables used successively as indices, where their cache miss rate should intuitively be much less than an access to a variable or complex data type in the same line of code.

Consider the snippet shown in Figure 5.3. In this example, the variable i is read on every line. At line 5, the likelihood of a miss for i is likely lower than that of the index to the array c. Using the raw approach previously described would result in an inflated miss count for the index variable i, while having lower than expected counts for each of the variables a, b, and c. We account for this by assigning weights to the variables within loops based on the frequency that they are accessed.

We define the source line where a miss occurred as S. We define the inner most loop that S is contained in as L. Within S, we define V to be the set of variables where a read occurs. For each variable v in V, equation 5.1 details the calculation of the raw weights,

$$v_{raw\_weight} = 1.0 - \left(\frac{\text{Number of lines v is read in } L}{\text{Number of lines in } L} \right)^2 \tag{5.1}$$

The raw weights for all variables will be below 1.0 as they will have at least one read within L. The weights will be redistributed favoring those variables with higher initial weights, with the sum of the final weights for all variables equaling the number of reads for the original line S. The equations for calculating the final weights are given below.

$$weight\_to\_redistribute = \sum(1.0 - v_{raw\_weight}) \tag{5.2}$$

We set a threshold for the raw weights. All variables above the threshold have their weights distributed according to their raw weight. Set V' is the set of variables with raw weights above the threshold. For all variables, v' within V',

$$threshold\_weights = \sum v'_{raw\_weight} \tag{5.3}$$

The adjusted weight is then calculated in equation 5.4.

$$v'_{adj\_weight} = \left((\frac{v'_{raw\_weight}}{threshold\_weights}) * _{weight\_to\_redistribute} \right) \tag{5.4}$$

Finally, the adjusted weight is added to the raw weight for our final weight. For those variables below the threshold, the final weight is equal to the raw weight.

$$v'_{final\_weight} = v'_{raw\_weight} + v'_{adj\_weight} \tag{5.5}$$

With the loop adjustment, the values for the weights from the code in Figure 5.2 are shown in Table 5.2. For this example, we assume a threshold of 0.75. Since

	Weights		
Variable	**Raw**	**Adjusted**	**Final**
x			
x[a]	.94	.52	1.46
x[b]	.94	.52	1.46
y			
y[a]	.94	.52	1.46
y[b]	.75	.42	1.17
a	.44	.00	.44
b	.00	.00	.00
Sum	4.0	2.0	6.0

Table 5.2: Weight used in calculating the loop adjustment

the threshold is being compared against raw weights, this threshold value marks the point where a variable is read in at most half of the lines in the loop. The threshold is represented by instantiating equation 5.1,

$$1.0 - (0.5^2) = 0.75$$

The different accesses for x and y are treated independently. We see that the final weight is equal to the number of memory locations (as distinguished through static analysis) that are read within the loop. When these final weights are applied to the original approximate misses (one miss from each source line) from before, we get the numbers shown in Table 5.3, with "LA" corresponding to the loop adjusted numbers.

	Approximate Misses							
	x[]		y[]		a		b	
	Raw	LA	Raw	LA	Raw	LA	Raw	LA
Line								
2	.25	.37	.25	.29	.25	.10	.25	.00
3	.00	.00	.50	.60	.25	.10	.25	.00
4	.50	.73	.00	.00	.00	.10	.50	.00
Sum	.75	1.10	.75	.89	.50	.30	.50	.00

Table 5.3: Cache misses assigned using 'Loop Compensation' technique

5.2.3 Skid Negation

The raw assignment and loop correction pass work best when skid is not a factor. However, when skid is taken into account, we need to provide a further pass to help negate the skid. The range and effect of skid manifests differently between architectures. We wanted our approach to be general, so that a minimal number of parameters would have to be specified to have our approach apply to a given architecture. By using an approach similar to that utilized by ProfileMe [18], we can generate a histogram for the distance between the true instruction and where the event based sample landed (the skid factor). Using this information, we can generate a probabilistic model for the effect of the skid. It should be noted that this histogram would not need to be generated per test program, but rather only once when wanting to run code on a new architecture. With this information, our approach need only two parameters: the mean and the variance. Using that information, we assign weights to the reads within the skid range.

An additional mapping is needed since our approach is source line based, and the skid distribution would be in terms of instructions. There are a few possibilities

for this mapping. The first would be to do a one-time linear scan of the instructions in the program and the associated valid source lines. With this information you can calculate the average number of instructions associated to a source line within the program. This could then be used to determine all possible "real" source lines the miss could correspond to. The second approach is more exact, and may be useful for programs with very large skid. This approach would look at the exact instruction given by the event and all of the instructions within the range of the skid distribution for that architecture. The associated source lines would be mapped accordingly.

In the results presented in this chapter, we experimented on architectures that utilize in-order execution and have more manageable skid factors. For this reason, we currently use the first mapping described in the above paragraph. For experiments on architectures with out-of-order execution, we will utilize the second style of mapping.

The process for calculating the skid negation is as follows. Formally, for set V which is all variables with reads within the possible distribution of the skid, for each v in V let p be probability assigned to the source line containing the read to v.

$$v\_raw\_skid\_val = p \tag{5.6}$$

We then take the sum of all the values in the skid distribution.

$$sum\_skid\_vals = \sum v_{raw\_skid\_val} \tag{5.7}$$

	Approximate Misses							
	x[]		y[]		a		b	
	Raw	SA	Raw	SA	Raw	SA	Raw	SA
Line								
2	0	.07	0	.07	0	.07	0	.07
3	0	.14	0	.14	0	.14	0	.14
4	.5	.07	0	0	0	0	.5	.07
Sum	.5	.28	0	.21	0	.21	.5	.28

Table 5.4: Cache misses assigned using 'Skid Negation' technique

We use that sum to normalize all weights, so the sum of all weights for a particular cache miss will equal one. For all v,

$$v\_final\_skid\_val = \left(\frac{v\_raw\_skid\_val}{sum\_skid\_vals}\right) \tag{5.8}$$

The final skid value is not used as a weight, which was the case with the loop compensation weights. It is used as a replacement to the raw approximate assignment values. In cases where there is skid, the skid negated value would be the base for passes such as the loop compensation. Table 5.4 shows the comparison between raw approximate assignment values versus skid adjusted values. We assume the cache miss was given as being on line 4, the skid distribution has a mean of line 3, with the variance being $\pm$ 1. We assign a probability of .5 to line 3 and .25 to line 2 and 4.

For both approximate miss methods, the sum of the attributed cache misses for the variables is 1. The raw methods assumes correct data for the attributed cache miss, which is why values are only present for line 4. The skid negation technique

assumes a probabilistic distribution over the source lines, which is why there are values assigned for lines 2, 3, and 4, even when the given line for the cache miss was line 4. Since this miss occurred within a loop, the next step would be to apply the loop compensation weights to the new values, which would see an increase in the attributed cache misses for the arrays, and a decrease for the scalars used to index them.

5.3 Experimental Results

For our experiments, we ran tests on three programs from the SPEC CPU2000 benchmark suite on an Intel Itanium 2 machine running Linux. The test programs and hardware configurations are the same that Buck used for his experiments. As we are comparing our approach against existing direct methods, it was important to have verified direct data centric results to compare against.

For our experimental programs, we gather data from three types of runs. All runs use event driven sampling to measure cache misses. For each type of run, we take the average values from five runs.

1. Sampling with skid free IP and precise effective address gathered using hardware support that negates the skid

2. Sampling with skid free IP without effective address gathered using hardware support that negates skid

3. Sampling with skid using generic hardware counter that does not negate skid

We use the values from the first type of run to perform a traditional direct data centric approach. This serves as a baseline to compare our approach against. For each graph in the results section, the given approximation approach will be compared against the direct approach.

We use the second type of run to perform our loop compensation adjustment. The lack of effective address information means we use our approach exclusively to assign the cache misses to the proper variables. The data from the final type of run suffers from skid, so we perform the loop compensation and the skid negation pass to assign miss responsibility to variables.

For the distribution of skid values, present in the third type of run, we use a probabilistic distribution similar to the one we used in our prior example. The Itanium used in these experiments utilizes in-order execution of instructions, meaning the skid is an artifact of the pipeline and is less severe compared to most out-of-order execution architectures. We set the skid distribution to be three source lines, with the mean being one source line prior to the one given. We use the same probabilistic distribution given in the example, .5 for the mean instruction and .25 for the other two (one of which is the source line given by the event driven sample).

For the graphs in the following sections, the direct approach will be shown as black bars on the left for each variable on each graph. The raw approximate adjustment is the first pass and is not meant to closely match the the direct measurements. The raw approach will be shown as white bars in the middle of the bars for each variable. The results of the loop adjustment, and when applicable, skid adjustment, are shown as gray bars to the far right. For each benchmark, the top

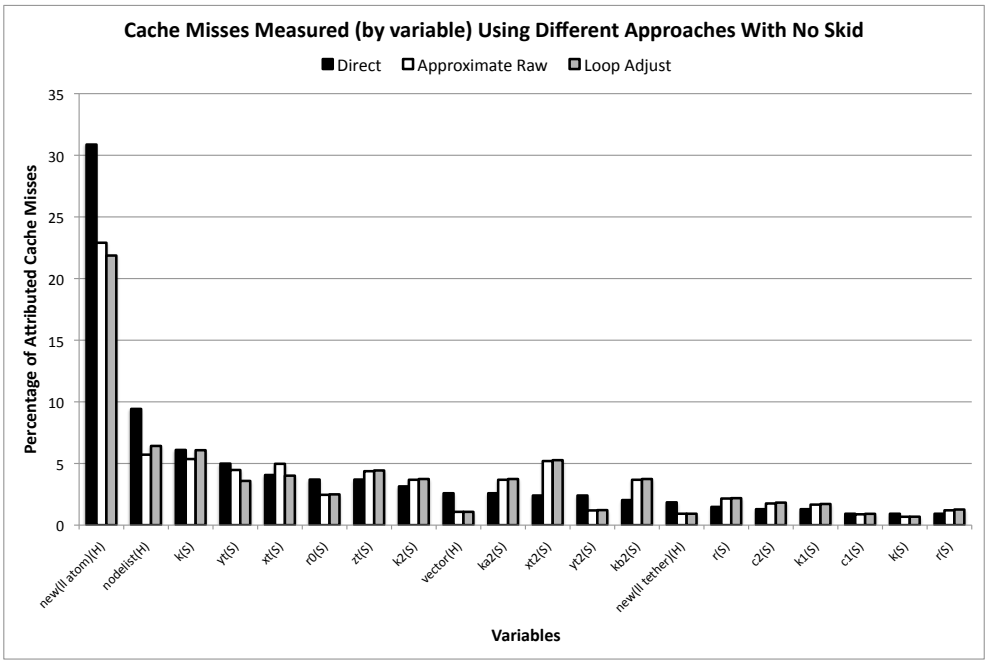


Figure 5.4: Cache misses for 'ammp' with skid negated

20 variables are shown, sorted in rank by the number of cache misses attributed by the direct method. The variable names are listed followed by the allocation method for the variable in parentheses. 'S' is for stack allocated variables, 'G' is for global, and 'H' is for heap.

5.3.1 188.ammp

The ammp benchmark is a C program that runs molecular dynamics on a protein-inhibitor complex that is embedded in water [65].

5.3.1.1 No Skid

We begin by applying our raw approximate assignment, shown in white bars, to the no-skid run. The results of this run are shown in Figure 5.4. The source lines

given for this run are guaranteed to be correct, so we compare our software only approach to the direct method, which was accomplished by comparing the effective address of the miss to a record of all the allocations and frees within the program. Our method compares favorably against the direct method. The heap allocated variable *new*, which is a node from a linked list, and the heap allocated variable *nodelist* suffer a decrease in the attributed cache misses, but the relative rank-order remains consistent. Since our approach is a software approximation, our goal is not an exact match to the 'true' data, but rather a light-weight mechanism to find the same hotspots that a direct approach (if possible on the hardware) would provide.

We then apply our loop compensation pass to the data, also shown in Figure 5.4 as the grey bars. This program has few loop nests. The loops within this program are very large with little use of loop iterators serving as indices into the arrays. Because of this, there are few artificially inflated outliers that the loop compensation pass would need to take care of, and the results are similar to that found by the the raw approximate approach.

5.3.1.2 Skid

The skid effect plays a major role in skewing the miss results as shown in Figure 5.5. The approximate raw results are significantly different than the true results given by the direct method. The skid adjustment pass does improve these results, but still struggles on some of the stack allocated scalars. This is due to the way the code is organized. Within the hotspots of the code is a series of memory

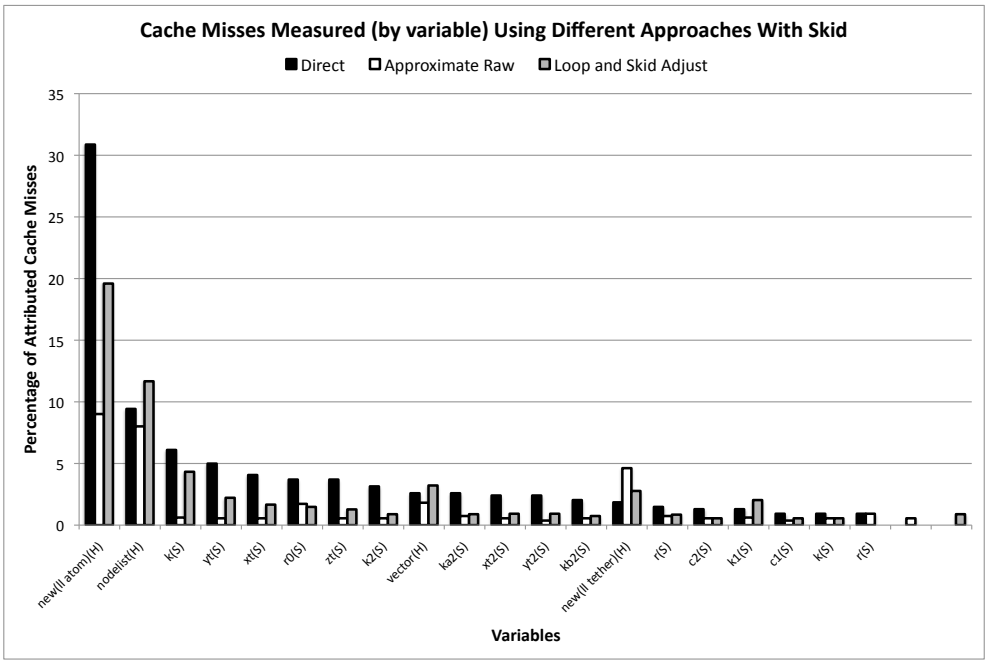


Figure 5.5: Cache misses for 'ammp' with skid

accesses using various local variables as the indices to the arrays and data structures. There is no locality in the use of the local variables and ten successive lines of code may use ten different local variables. For this reason, the skid makes a significant impact on the accuracy of the results. Because our approach tries to approximate the skid, we are able to correct the behavior and improve the results. For the heap allocated variables, locality plays a bigger factor and our results are more accurate.

5.3.2 173.equake

The equake benchmark is a C program that simulates the propagation of waves in large, heterogeneous valleys [8].

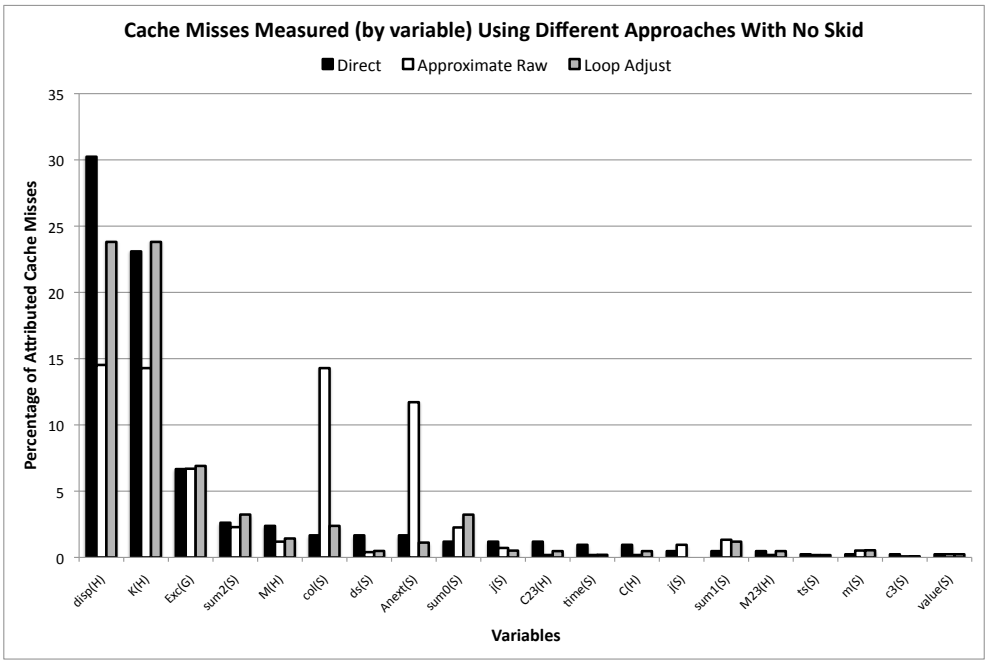


Figure 5.6: Cache misses for 'equake' with skid negated

5.3.2.1 No Skid

The results of the raw approximate assignment and loop compensation are shown in Figure 5.6. For this benchmark, there are outliers based on misappropriated misses to scalars. The loop compensation pass takes care of many of these discrepancies. After applying the loop compensation the main inaccuracy, as compared to the direct method, is the lower than expected values for variable *disp*. The reason for this is that *disp* and k are present together on the same lines for most of their reads and have the same number of unique reads on each one of these lines. In terms of the lines where the misses actually occur, their signatures are virtually identical in regards to our analysis. This leads to a close to expected value for k, but a lower value for *disp*.

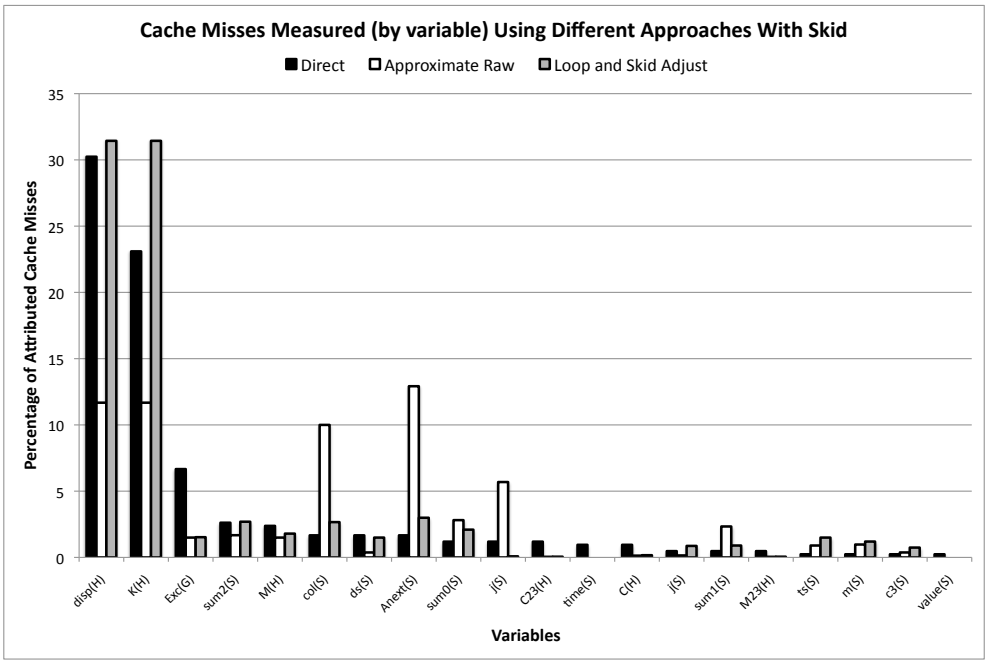


Figure 5.7: Cache misses for 'equake' with skid

5.3.2.2 Skid

The prior runs benefited from having the correct source line associated with the cache miss. The raw assignments with the skid active give us the data in Figure 5.7. Due to the skid factor, the outliers are more apparent and the variable assignments are much different than the baseline direct data method. By applying our software skid negation, we achieve more reasonable data. The results of the skid adjustment (with combined loop adjustment) are also shown in Figure 5.7, with the adjustments represented by the gray bars.

The two main variables, *disp* and k, are once again linked together and at the top of the list of the appropriated misses. The skid adjustment helped to improve the results for these two variables. This is because the source line wrongly attributed for many cache misses was one line ahead of a statement that performed multiple reads.

The attributed misses for variable *Exc* was low before the skid adjustment and the value remained low after the skid adjustment. This was due to the misattributed source line being directly after a call in which a read to *Exc* was the comparison operation before the return. At this time, our analysis does not move backwards through calls, so we were unable to redistribute values to that variable.

5.3.3 179.art

The ART 2 benchmark is a C program that uses neural network models to recognize objects in a thermal image. [16]

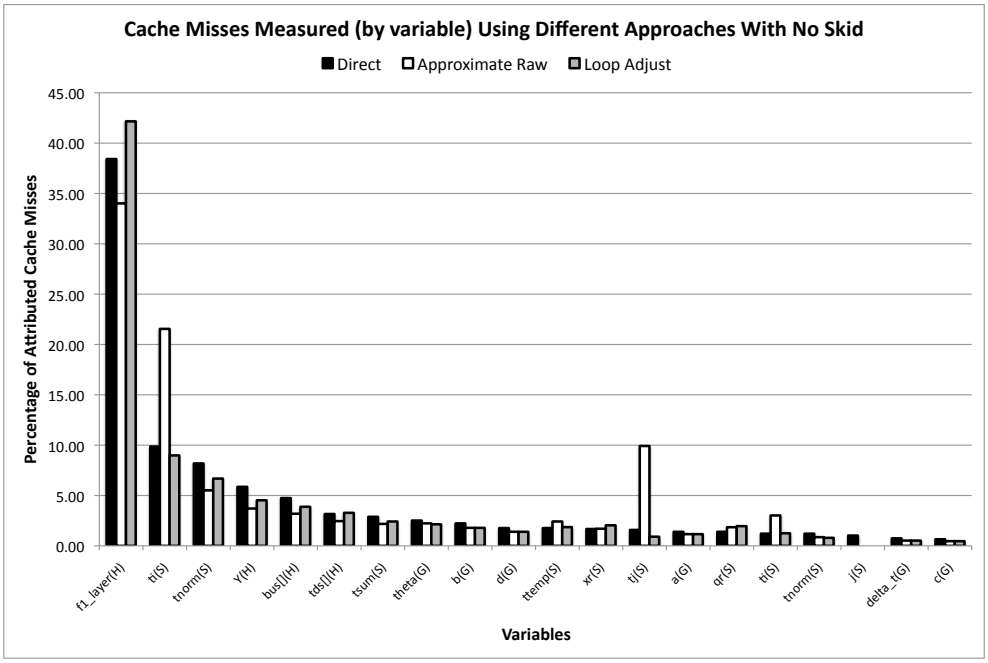


Figure 5.8: Cache misses for 'art' with skid negated

	Rank Order		
Variables	**Direct**	**Skid Raw**	**Skid Adjusted**
f1_layer(H)	1	2	1
ti (S)	2	1	2
tnorm(S)	3	19	3
Y(H)	4	15	4
bus[](H)	5	9	14
tds[](H)	6	8	6
tsum(S)	7	16	7
theta(G)	8	10	8
b(G)	9	12	11
d(G)	10	11	12

Table 5.5: Rank-Order Comparison for Top 10 Miss Causing Variables

5.3.3.1 No Skid

We first examine our raw approach versus the direct data method, shown in Figure 5.8. Our initial method performs comparably against the direct method with the exception of the outliers for stack allocated variables *ti*, *tj*, and *ti*. These are all loop iterator variables, and are the motivation for the loop adjustment pass. The loop adjustment pass eliminates the outliers. For this program, our method lines up almost perfectly with the direct method.

5.3.3.2 Skid

The raw approximate values for the skid run are shown in Figure 5.9. The skid factor results in diminished values for most of the top 10 variables found by the direct method. Our skid adjusted approach gives us a rank-order approximately the same as the direct method. Table 5.5 shows the rank-order for the top ten variables (as given by the direct data approach).

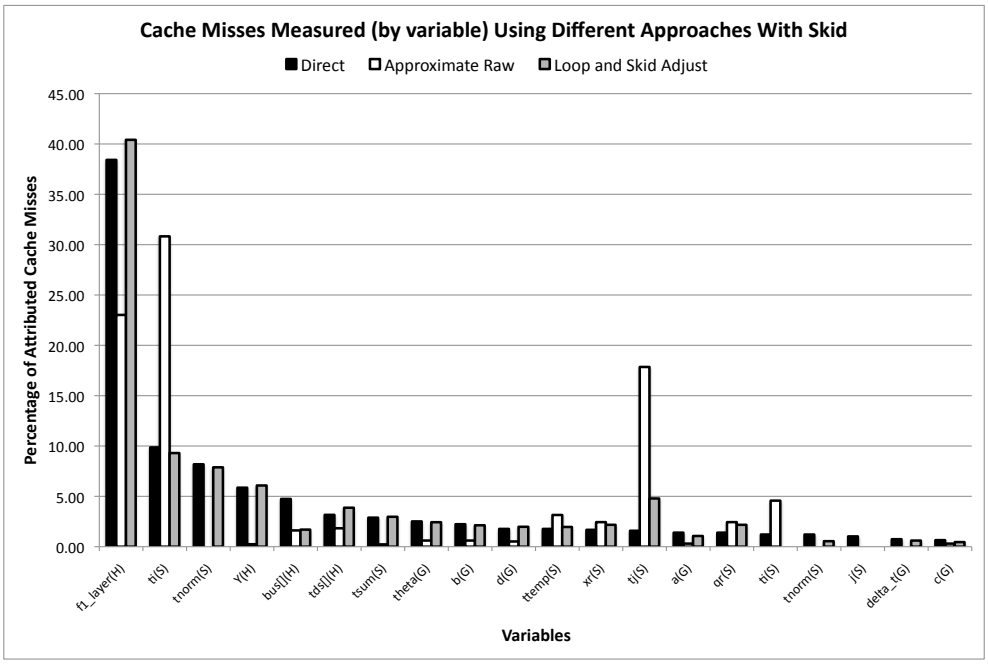


Figure 5.9: Cache misses for 'art' with skid

The low number of misses assigned to *bus* is due to the reads for that variable occurring in the middle of a set of conditionals, with the misappropriated instruction occurring immediately after the block for the final 'else.' A solution to this problem would be assigning probabilistic weights to different regions in the conditional block based on generic profiling numbers, but that is not present in our current analysis.

5.3.4 Correlation to Direct Measurement

Our approaches utilize approximation techniques. We have discussed the comparisons of rank order between the direct method, which serves as the ground truth, and our approximation. We can also examine the correlation coefficient between the cache misses assigned by direct measurement and those assigned by our technique(s).

Table 5.6 shows the correlation coefficient (PPMCC) [54] between our ap-

	Correlation Coefficient			
	No Skid		Skid	
Benchmark	App. Raw	Loop Adjust	App. Raw	Loop & Skid Adjust
ammp	.97	.97	.79	.95
equake	.91	.99	.61	.99
art	.72	.98	.65	.98

Table 5.6: Correlation Coefficient of our approaches versus Direct Measurement

proaches and the direct measurement method. The approximate raw technique is highly correlated in those programs where loops are not a factor. This is due to the locality issues and the misattributed misses to loop index variables. The *art* program is especially guilty of these misattributions and has a lower correlation coefficient as a result. When the loop adjustment is applied, all three benchmarks have very high correlation for the misses attributed to the variables in our approach versus the direct method.

The raw skid correlations are lower across the board. There is still a strong positive correlation due to the locality of variable accesses within the program, but is not a high enough correlation to be useful as an approximation technique. When the loop and skid adjust passes are applied, however, we get the very high correlation numbers that we saw in the negated skid runs. This result shows that our skid negation techniques are a valid approach on hardware where direct measurement is not possible.

5.4 Summary

In this chapter, we have discussed a technique for approximating cache miss totals for variables using only static analysis and runtime data gathered from event driven sampling utilizing generic hardware counters. Existing techniques resolve these misses by matching the effective address to a maintained list of allocated memory by monitoring allocations and frees. Our approach allows this type of analysis to run on systems without dedicated hardware support that provides exact effective address and IP information unaffected by skid. It also removes the need to maintain the allocation list. Our approach is meant to be a supplementary method to existing techniques, in cases where the hardware does not allow the existing types of analysis to be run or allocation monitoring adds too much overhead.

Chapter 6

Future Work

6.1 Large Scale Data Presentation

We currently separate how we internally store drill down data for small (32 processes or less) and large runs. For small runs, we input all of the data into our GUI explicitly so drill down operations can be done without having to pull external data. Conversely, for large runs, we utilize map-reduce operations to aggregate data to alleviate storage concerns effectively eliminating some of the drill down data. We want to examine ways to compactly store runtime information so we can maintain a small storage footprint while still being able to provide extensive drill down information for large runs.

6.2 Blame at the instruction level

One area of future work would involve doing the mappings at the instruction level to compare the accuracy. This would have the advantage of not having to divide the sample over all of the reads on the mapped source line as is the case in our current approach. However, this would involve having reliable alias analysis and other static analysis at the machine code level for any platform we wished to support.

6.3 Blame combined with autotuning

The next step for this work is continuing to utilize the data to improve program performance. Our work is able to localize the amount of time spent in populating the data for a given data structure. Many of the variables shown in our results have tunable parameters that affect how much computation goes into calculating the data for that variable. These tunable parameters range from the communication patterns used for distributed data structures to the underlying data structures that represent the variable such as whether to use a sparse or dense matrix. Our current work involves taking the most blamed variables and tweaking those tunable parameters to try to increase program performance. Our future work will entail a two pass approach, where we use our analysis to figure out relevant problem variables to reduce the state space and then use a combination approach of our analysis and autotuning to find optimal data structures to solve the problem.

6.4 Runtime and Post-Processing Optimizations

We have discussed how our runtime overhead can be reduced by creating a call context tree instead of storing each individual call trace. We can also use similar data structures to improve our post-processing. We currently apply transfer functions to individual samples at every frame in the call context. By storing past transfer functions that we have applied to various calling contexts, we can alleviate the need to apply transfer functions to a sample if they have already been applied on previous samples.

6.5 Skid approximation algorithms on out of order architectures

The main area of future work for the approximate data centric approach is to apply the techniques to other architectures that have skid-negating hardware, specifically those platforms with out of order execution. An increased distribution of skid values creates backtracking problems in terms of control flow. We have already encountered issues with conditionals and function calls. The greater the skid factor, the more considerations about control flow become a factor. Expanding our approach to other platforms would allow us to further test the validity of our technique across different degrees of skid and cache replacement policies, both of which may affect how our approach will perform.

Chapter 7

Conclusions

In this dissertation, we introduced new techniques for performing data centric performance profiling. These techniques allow developers to gather information about program performance in ways that were not possible before. In the case of variable blame, performance information can be mapped to program variables in a unique manner that can not be duplicated by existing code centric techniques. In the case of our approximate data centric technique, existing direct measurement approaches can be approximated on architectures that do not have the hardware features that existing approaches require. The results presented in this document show that our techniques succeed in calculating new forms of data centric analysis that provide useful information for performance profiling.

To evaluate the effectiveness of our variable blame analysis, we established a metric to determine the uniqueness of the data presented in comparison to what data could be recorded by classic code centric analysis techniques. We then ran our blame analysis on real systems to increase program understanding of the profiled programs. We were able to find multiple variables with unique blame values (as defined by our uniqueness metric) for each program, and showed how these variables held useful aggregates of operations occurring within the program. In the case of PFLOTRAN, we used this information to better program performance. We did this by modifying

input parameters for those variables deemed interesting by our blame metric (based on the calculated computation time to populate these variables).

For our approximate data centric technique, our concern was not to find unique data in comparison to existing techniques, but rather to approximate those techniques. We examined established data centric techniques that had limitations on which architectures they could be run on based on hardware counter support. The evaluation of the effectiveness of our technique was done by comparing our results, gathered using only basic hardware support, with the results from those techniques that required more dedicated hardware. We showed that our technique is able to maintain the approximate rank order as the existing technique with a range of .97-.99 correlation coefficients with no skid and a .95-.99 correlation range with skid.

We also provide a formal definition for variable blame and details about the graphical representation used within our internal blame implementation. Our system mixes pre-run, runtime, and post-run information before presenting the final data to the user. Much of our blame tool is interconnecting components that can be used for other forms of analysis. Our approximate data centric had minimal overlaps with variable blame in regards to how the metric was computed. However, it exclusively used the same internal graph representation that was utilized by the blame computation, as well as the same runtime engine, and the same GUI used by the blame tool. We believe that other types of data centric analysis tools can be built based on the same concepts introduced in this document. Mainly, the use of pre-run static analysis to expose fundamental program properties and binding that information to the minimum amount of necessary data gathered at runtime.

Appendix A

Raw LLVM Code

```
define void @bar(i32* %x1, i32* %y2) nounwind  {
entry:
   %x_addr = alloca i32* ; <i32**> [#uses=3]
   %y_addr = alloca i32* ; <i32**> [#uses=3]
   %loopC = alloca i32 ; <i32*> [#uses=6]
   %i = alloca i32 ; <i32*> [#uses=5]
   %"alloca point" = bitcast i32 0 to i32 ; <i32> [#uses=0]
   %x = bitcast i32** %x_addr to {  }* ; <{  }*> [#uses=1]
   store i32* %x1, i32** %x_addr
   %y = bitcast i32** %y_addr to {  }* ; <{  }*> [#uses=1]
   store i32* %y2, i32** %y_addr
   %loopC3 = bitcast i32* %loopC to {  }* ; <{  }*> [#uses=1]
   %i4 = bitcast i32* %i to {  }* ; <{  }*> [#uses=1]
   store i32 0, i32* %loopC, align 4
   store i32 0, i32* %i, align 4
   br label %bb8

bb: ; preds = %bb8, %0
   %tmp = load i32* %loopC, align 4 ; <i32> [#uses=1]
   %tmp5 = add i32 %tmp, 1 ; <i32> [#uses=1]
   store i32 %tmp5, i32* %loopC, align 4
   %tmp6 = load i32* %i, align 4 ; <i32> [#uses=1]
   %tmp7 = add i32 %tmp6, 1 ; <i32> [#uses=1]
   store i32 %tmp7, i32* %i, align 4
   br label %bb8

bb8: ; preds = %bb, %entry
   %tmp9 = load i32* %i, align 4 ; <i32> [#uses=1]
   %tmp10 = icmp sle i32 %tmp9, 9 ; <i1> [#uses=1]
   %tmp1011 = zext i1 %tmp10 to i8 ; <i8> [#uses=1]
   %toBool = icmp ne i8 %tmp1011, 0 ; <i1> [#uses=1]
   br i1 %toBool, label %bb, label %bb12
   br label %bb12

bb12: ; preds = %1, %bb8
   %tmp13 = load i32* %loopC, align 4 ; <i32> [#uses=1]
   %tmp14 = and i32 %tmp13, 1 ; <i32> [#uses=1]
   %tmp1415 = trunc i32 %tmp14 to i8 ; <i8> [#uses=1]
   %toBool16 = icmp ne i8 %tmp1415, 0 ; <i1> [#uses=1]
   br i1 %toBool16, label %bb17, label %bb21
   br label %bb17
```

Figure A.1: LLVM IR for ‘bar’

```
bb12: ; preds = %1, %bb8
   %tmp13 = load i32* %loopC, align 4 ; <i32> [#uses=1]
   %tmp14 = and i32 %tmp13, 1 ; <i32> [#uses=1]
   %tmp1415 = trunc i32 %tmp14 to i8 ; <i8> [#uses=1]
   %toBool16 = icmp ne i8 %tmp1415, 0 ; <i1> [#uses=1]
   br i1 %toBool16, label %bb17, label %bb21
   br label %bb17


bb17: ; preds = %2, %bb12
   %tmp18 = load i32** %x_addr, align 4 ; <i32*> [#uses=1]
   %tmp19 = getelementptr i32* %tmp18, i32 0 ; <i32*> [#uses=1]
   %tmp20 = load i32* %loopC, align 4 ; <i32> [#uses=1]
   store i32 %tmp20, i32* %tmp19, align 4
   br label %bb25
   br label %bb21

bb21: ; preds = %3, %bb12
   %tmp22 = load i32** %y_addr, align 4 ; <i32*> [#uses=1]
   %tmp23 = getelementptr i32* %tmp22, i32 0 ; <i32*> [#uses=1]
   %tmp24 = load i32* %loopC, align 4 ; <i32> [#uses=1]
   store i32 %tmp24, i32* %tmp23, align 4
   br label %bb25

bb25: ; preds = %bb21, %bb17
   br label %return

return: ; preds = %bb25
   ret void
}
```

Figure A.2: LLVM IR for ‘bar’ (continued)

```
define void @foo() nounwind  {
entry:
     %se = alloca %struct.StructEx ; <%struct.StructEx*> [#uses=5]
    %"alloca point" = bitcast i32 0 to i32 ; <i32> [#uses=0]
    %se1 = bitcast %struct.StructEx* %se to {  }* ; <{  }*> [#uses=1]
    %tmp = call i8* @malloc( i32 40 ) nounwind  ; <i8*> [#uses=1]
    %tmp2 = bitcast i8* %tmp to i32* ; <i32*> [#uses=1]
    %tmp3 = getelementptr %struct.StructEx* %se, i32 0, i32 0   ; <i32**> [#uses=1]
    store i32* %tmp2, i32** %tmp3, align 4
    %tmp4 = call i8* @malloc( i32 40 ) nounwind  ; <i8*> [#uses=1]
    %tmp45 = bitcast i8* %tmp4 to i32* ; <i32*> [#uses=1]
    %tmp6 = getelementptr %struct.StructEx* %se, i32 0, i32; <i32**> [#uses=1]
    store i32* %tmp45, i32** %tmp6, align 4
    %tmp7 = getelementptr %struct.StructEx* %se, i32 0, i32 1 ; <i32**> [#uses=1]
    %tmp8 = load i32** %tmp7, align 4 ; <i32*> [#uses=1]
    %tmp9 = getelementptr %struct.StructEx* %se, i32 0, i32 0 ; <i32**> [#uses=1]
    %tmp10 = load i32** %tmp9, align 4 ; <i32*> [#uses=1]
    call void @bar( i32* %tmp10, i32* %tmp8 ) nounwind
    br label %return

return: ; preds = %entry
    ret void
}
```

Figure A.3: LLVM IR for ‘foo’

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19:137–, March 1976.

[3] 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.

[4] Darren C. Atkinson, Markus Mock, Craig Chambers, and Susan J. Eggers. Program Slicing Using Dynamic Points-to Data. In *ACM SIGSOFT 10th Symposium on the Foundations of Software Engineering (FSE*, pages 71–80, 2002.

[5] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[6] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[7] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[8] Hesheng Bao, Jacobo Bielak, Omar Ghattas, David R. O'Hallaron, Loukas F. Kallivokas, Jonathan Richard Shewchuk, and Jifeng Xu. Earthquake Ground Motion Modeling on Parallel Computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, November 1996.

[9] Erik Berg and Erik Hagersten. Statcache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *In Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2004.

[10] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening Multigrid on Distributed Memory Machines. *SIAM J. Sci. Comput.*, 21(5):1823–1834, 2000.

[11] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *In Proceedings of Supercomputing*, 2000.

[12] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[13] Bryan R. Buck. Data Centric Cache Measurement on the Intel Itanium 2 Processor. In *In: Proceedings of SuperComputing. (2004*, 2004.

[14] Bryan R. Buck. *Data Centric Cache Measurement Using Hardware and Software Instrumentation*. PhD thesis, University of Maryland, 2004.

[15] Bug Cluster. `https://wiki.umiacs.umd.edu/umiacs/index.php/BugCluster/`.

[16] Gail A. Carpenter and Stephen Grossberg. Art 2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns. *Appl. Opt.*, 26(23):4919–4930, Dec 1987.

[17] Carver configuration. `http://www.nersc.gov/users/computational-systems/carver/configuration/`.

[18] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[19] Alain Deutsch. On the complexity of escape analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 358–371, New York, NY, USA, 1997. ACM.

[20] P. J. Drongowski. "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors". Technical report, Advanced Micro Devices, Inc., 2007.

[21] FFP_SPARSE. `http://people.scs.fsu.edu/~burkardt/cpp_src/ffp_sparse/`.

[22] Robert Bruce Findler. *Behavioral Software Contracts*. PhD thesis, RiceUniversity, 2002.

[23] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead Call Path Profiling of Unmodified, Optimized Code. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM.

[24] M. Gerndt, B. Mohr, M. Pantano, and F. Wolf. Automatic Performance Analysis for Cray T3E. In *Proc. of the 7th Workshop on Compilers for Parallel Computers (CPC'98)*, pages 69–78, University of Linköping, Sweden, June-July 1998.

[25] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[26] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An Execution Profiler for Modular Programs. *Softw., Pract. Exper.*, 13(8):671–685, 1983.

[27] Axel Gross. Evaluation of Dynamic Points-to Analysis, 2004.

[28] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic Inference of Abstract Types. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, New York, NY, USA, 2006. ACM.

[29] Jeffrey K. Hollingsworth. *Finding Bottlenecks in Large Scale Parallel Programs*. PhD thesis, University of Wisconsin-Madison, 1994.

[30] HPL. `http://www.netlib.org/benchmark/hpl/`.

[31] Intel. Intel performance tuning utility website. `http://softwarecommunity.intel.com/articles/eng/1437.htm`.

[32] Intel. Intel VTune. `http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm`.

[33] Intel. *Intel Itanium 2 Processor Reference Manual*. Intel, May 2004.

[34] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2*. Intel, June 2010.

[35] Franois Irigoin, Pierre Jouvelot, and Rmi Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *ICS'91*, pages 244–251, 1991.

[36] R. Bruce Irvin. *Performance Measurement Tools for High-Level Parallel Programming Languages*. PhD thesis, University of Wisconsin-Madison, 1995.

[37] R. Bruce Irvin and Barton P. Miller. Mapping Performance Data for High-Level and Data Views of Parallel Program Performance. In *International Conference on Supercomputing*, pages 69–77, 1996.

[38] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *In Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[39] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04)*, 2004.

[40] Xu Liu and John Mellor-Crummey. Pinpointing Data Locality Problems Using Data-centric Analysis. In *International Symposium on Code Generation and Optimization (CGO11)*, 2011.

[41] LLVM. LLVM Alias Analysis Infrastructure . `http://llvm.org/docs/AliasAnalysis.html`.

[42] LLVM. The Often Misunderstood GEP Instruction . `http://llvm.org/docs/GetElementPtr.html`.

[43] Allen D. Malony. Tools for Parallel Computing: A Performance Evaluation Perspective. In Jacek Bazewicz, Denis Trystram, Klaus Ecker, and Brigitte Plateau, editors, *Handbook on Parallel and Distributed Processing*, chapter VII, pages 342–363. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[44] Stephen McCamant and Michael D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205, Tucson, AZ, USA, June 9–11, 2008.

[45] Collin Mccurdy and Jeffrey Vetter. Memphis: Finding and Fixing Numa-Related Performance Problems on Multi-Core Platforms. In *In Proceedings of ISPASS*, 2010.

[46] John M. Mellor-Crummey, Robert J. Fowler, and David B. Whalley. Tools for Application-Oriented Performance Tuning. In *International Conference on Supercomputing*, pages 154–165, 2001.

[47] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[48] Pedro Mindlin, Jose R. Brunheroto, Luiz DeRose, and Jose E. Moreira. Obtaining Hardware Performance Metrics for the BlueGene/L Supercomputer. In *Euro-Par '03: Proceedings of the 9th International Euro-Par Conference on Parallel Processing*, pages 21–32, 2003.

[49] Markus Mock, Markus Mock, Manuvir Das, Manuvir Das, Craig Chambers, Craig Chambers, Susan J. Eggers, and Susan J. Eggers. Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization. In *In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72. ACM Press, 2001.

[50] Paraver. `http://www.cepba.upc.edu/paraver/`.

[51] Young Gil Park and Benjamin Goldberg. Escape Analysis on Lists. In *In Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 116–127. ACM, 1992.

[52] PFLOTRAN. `http://ees.lanl.gov/pflotran/`.

[53] QUAD. `http://people.sc.fsu.edu/~burkardt/c_src/quad_mpi/`.

[54] J. L. Rodgers and W. A. Nicewander. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician*, 42:59–66, 1988.

[55] Luiz De Rose, Ying Zhang, and Daniel A. Reed. SvPablo: A Multi-language Performance Analysis System. *Lecture Notes in Computer Science*, 1469:352–355, 1998.

[56] SGI Technical Publications. *SpeedShop User's Guide*.

[57] Sameer Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, 2001.

[58] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[59] SMG2000. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/`.

[60] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary Analysis for Measurement and Attribution of Program Performance. *SIGPLAN Not.*, 44:441–452, June 2009.

[61] The GNU C Library. `http://www.gnu.org/s/hello/manual/libc/index.html`.

[62] Univ. of Maryland, Univ. of Wisconsin. *StackWalker API Manual*, 0.6b edition, 2007.

[63] Vampir. `http://www.vampir.eu/index.html`.

[64] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.

[65] Irene T. Weber and Robert W. Harrison. Molecular Mechanics Analysis of Drug-Resistant Mutants of HIV Protease. *Protein Engineering*, 12(6):469–474, 1999.

[66] Mark Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[67] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1995. ACM.

[68] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.