LANGUAGE EXTENSIONS FOR PERFORMANCE-ORIENTED PROGRAMMING

BY

JOEL STANLEY

B.S., University of Portland, 2001

THESIS

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

Urbana, Illinois

For Joanna, my oasis in the Desert of the Real.

Abstract

Modern software development practices lack portable, precise and powerful mechanisms for describing performance properties of application code. Traditional approaches rely almost solely on performance instrumentation libraries, which have significant drawbacks in certain types (e.g., adaptive) of applications, present the end user with integration challenges and complex APIs, and often pose portability problems of their own. This thesis proposes a small set of C-like language extensions that facilitate the treatment of performance properties as intrinsic properties of application code. The proposed language extensions allow the application developer to encode performance expectations, gather and aggregate various types of performance information, and more, all at the language level. Furthermore, this thesis demonstrates many novel compiler implementation techniques that make the the presented approach possible with an arbitrary (third-party) compiler, and that minimize performance perturbation by enabling compiler optimizations that are commonly inhibited by traditional approaches. This thesis describes the fundamental contribution of language-level performance properties, the language extensions themselves, the implementation of the compilation and runtime system, together with a standard library of widely-used metrics, and demonstrates the role that the extensions and compilation system can play in describing the performance-oriented aspects of both a production-quality raytracing application and a long-running adaptive server code.

Acknowledgments

Within the academic and professional realm, I would like to thank my advisor Vikram Adve for his compiler expertise and for the work environment that he fostered. I would also like to thank Chris Lattner for all of the technical support and fast bugfixes that he provided for the excellent LLVM compiler infrastructure. Much thanks to Brian Ensink for being an absolutely fantastic co-worker and a great guy to bounce ideas off of. Finally, heartfelt thanks to Lewis Lum for mentoring me and for showing me what I was capable of.

On a personal level, I would like to first acknowledge the continual and loving support of my wife, Joanna, who has been a constant source of inspiration to me. Also, many thanks to my friends from Portland (you all know who you are), without whom I was likely to have been eaten by a grue. My parents also deserve a great deal of thanks for their continual support of me and my pursuit of a solid education.

Finally, my sanity would have been considerably more difficult to maintain throughout this process without the musical and literary influences of: Switchblade Symphony, Rasputina, Tool, Eridu Arcane, Händel, Arvo Pärt, T.S. Eliot, H.P. Lovecraft, Billy Collins, Paul Tillich, Martin Heidegger, Robert Solomon, and many others.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Performance instrumentation is the act of gathering data about a program's performance during execution without altering the goals of the underlying computation. These performance data are gathered at runtime, usually by evaluating performance metrics, which are functions that vary with respect to time and describe some feature of a program during execution. Because high performance is an important goal for many classes of software applications, the quality of these data, the specific ways in which they are gathered, and their efficacy in locating performance bottlenecks are of utmost concern.

Numerous performance instrumentation methodologies (e.g., [32, 26, 20, 21, 19, 18]) have been proposed, and many techniques (e.g., [33, 3, 36, 7]) for instrumenting programs have been suggested. Despite this, however, modern software development techniques fail to provide a way for applications to express their own performance requirements, expectations, or metrics as features of the application code (these are collectively referred to as the performance properties of the code). Instead, performance goals for high performance applications are typically met by an iterative process of repeated data gathering, manual application tuning, and (usually) offline data analysis performed by programmers or external tools. This entire process is necessarily ad hoc because it cannot be described in the application code, and is closely tied to the capabilities and interfaces of tools on each platform.

The broad goal of this thesis is to address problems with typical performance instrumentation methodologies by developing and demonstrating relatively simple language extensions that allow arbitrary performance properties to be *expressed as basic features of a program.* In particular, the extensions must permit a program to refer explicitly to its own performance properties within the code, obtain runtime values for certain properties during program execution, and use these values either for performance diagnosis or runtime adaptation. Most importantly, the sequence of performance observations (though of course not the observed values) should be precisely reproducible (i.e., executed in the same order in any execution of the program on any machine, just as the observable read and write operations in a program are precisely reproducible). Also, the performance properties must be able to remain in the program code while incurring negligible runtime overhead for production environments (i.e., it should not be necessary to resort to preprocessor mechanisms in order to enable or disable performance instrumentation).

A second and related goal of this thesis is to develop compiler implementation strategies that are *practical* to incorporate into real-world development environments, which must work with arbitrary (third-party) compilers, development environments, and performance monitoring tools and mechanisms. The question of *how* to achieve such a practical solution has remained unaddressed in the literature.

The next section addresses some problems with the traditional approaches. Section 1.2 goes on to discuss potential benefits offered by a language-based approach like the one proposed in this thesis. Section 1.3 provides a high-level overview of the thesis and summarizes its contributions. Finally, an outline of the thesis is provided in Section 1.4.

1.1 Drawbacks of Traditional Performance Instrumentation Methodologies

Traditional performance instrumentation methodologies have a number of significant drawbacks that are worth illuminating. This section presents a brief discussion of the traditional approaches and their drawbacks, touching on by-hand instrumentation techniques, common tool-based approaches, and mature performance instrumentation systems.

As noted in the previous section, current instrumentation techniques are often ad hoc. A typical simple instrumentation scenario involves manual instrumentation (e.g., counter variables, debugging output statements, manually-placed timers, and so on) of application code. This approach has the obvious disadvantage that it is tedious, usually requires preprocessor mechanisms to enable or disable instrumentation, and can easily fail to elucidate any issues that could be exposed via a more macroscopic evaluation of the program's performance. Although sophisticated instrumentation libraries (e.g., [11]) may help, we believe that this thesis offers numerous advantages over a library approach (see Section 1.2).

More gratifying instrumentation approaches include profiling or tracing. Although these are often successful in identifying portions of code that should be optimized, they are very general-purpose and can usually only provide coarse-grained factual information (e.g., that the most execution time was spent *in* a particular routine), and not much information that is relevant to a particular application domain (e.g., that the most execution time was spent in a particular routine *because* "too many" server requests occurred during execution). Usually, only a few complex metrics (if any) are exposed to application developers by these approaches. Although notable modern profiling systems (e.g., [35]) continue to get better, this thesis proposes an alternate instrumentation strategy altogether.

A number of very sophisticated performance instrumentation tools, such as Pablo [32] and Paradyn [26], are much better. They provide rich sets of metrics within the context of scalable analysis frameworks, which allow both high- and low-level performance instrumentation to occur. Paradyn, e.g., adopts a complex search model [18] that can used for automatically detecting performance bottlenecks in parallel codes, while Pablo provides a diverse set of statistical analyses that can be applied to gathered metric data. Both systems have significant drawbacks, however. For example, Pablo performs most of its data analyses (e.g., correlating performance data with their potential causes) offline, which means that the application cannot respond (i.e., perform runtime adaptations) based on the computed analysis results. As another example, Paradyn employs a technique called dynamic instrumentation which allows for on-the-fly insertion of instrumentation code. Unfortunately, the granularity of potentially instrumentable points is very coarse (i.e., only procedure entry, exit, and call sites are supported), which limits the kinds of instrumentation that can occur.

Neither of these exemplary systems permit the specification of performance properties in the source code of the application itself, nor do they allow the collected performance data to be examined by the instrumented program at runtime. We believe these to be significant limitations of these systems, and that these systems are representative of the current approaches to performance instrumentation. Ergo, this thesis presents an alternative, language-based methodology in order to address these issues.

1.2 Potential Benefits of a Language-Based Methodology

Language extensions for performance properties provide several useful capabilities (some are novel, but all are unified) which are powerful but rarely used in current software practices:

1. *Performance self-monitoring with dynamic control*: Language extensions would make it natural and straightforward to write performance diagnosis code within an application itself, and in particular to compute and report *application-specific* performance metrics during program execution. Furthermore, dynamic control allows the application to specify the addition or removal of instrumentation at runtime. To date, self-monitoring of performance is used only in very limited ways by most high performance applications (e.g., to report simple cumulative metrics such as execution time or MFlops).

2. *Runtime adaptation*: Many classes of applications (especially, distributed applications such as parallel Grid codes or distributed multimedia codes) must use runtime adaptation to achieve desired performance or meet Quality-of-Service requirements under changing runtime conditions. Language extensions for performance feedback would allow such runtime adaptation to observe and use program performance properties relatively easily via reproducible performance monitoring operations.

3. *Performance Assertions*: Programs could easily include performance assertions that express the expectations of the software designer and generate runtime events (e.g., exceptions or callbacks) when an expectation is not met. Such assertions would become simple expressions anywhere in the program. A few papers have proposed the idea of performance assertions (e.g., expressed as significant comments in the code) [38, 29]. We achieve a similar goal but through more general mechanisms, as discussed in Chapter 2. Performance assertions can also create ideal locations in the code for an application designer to place dynamic instrumentation

control directives (e.g., adding a new metric to compute at an instrumentation site whenever a particular performance assertion is violated) .

4. *Instrumentation-aware compilation systems*: One key aspect of encoding performance properties at the language level is that the compilation system can be aware of all code that has been instrumented by the user, as well as the precise nature of that instrumentation. This opens the door to program analyses and optimizations that directly relate to the ways in which instrumentation is deployed in the target application. For example, instrumentation-aware compilation systems can provide the basic mechanisms necessary for semi-automatic performance prediction (see next bullet). As an example closely related to semi-automatic performance prediction, consider the problem commonly faced by application programmers and performance tuners when attempting to analyze pipeline behavior. Such analysis is often difficult because pipeline performance problems are typically hard to instrument externally. However, compilers often optimistically model pipelines. With performance properties, a compiler could intercept a pipeline metric (e.g., pipeline bubbles) and analyze the late-stage generated code (e.g., code contained in a dynamic trace).

5. *Semi-automatic Performance Prediction*: A fundamental obstacle to the widespread use of application performance modeling is that real-world application development teams often lack the expertise to model low-level system features such as processor pipelines or memory hierarchies, while compiler-based modeling techniques are unable to construct models for application-specific properties and metrics.

   An attractive long-term solution is a semi-automatic approach that integrates compiler-generated models for complex system features with manually generated (i.e., programmed) models for application-level behavior, thus "bridging the gap" between the application domain and the low-level system features. Language extensions for performance properties *provide exactly the mechanisms required to make this possible*: they can tell the compiler what models to generate, and they provide primitives that application code could use to incorporate the results of these models into application-specific expressions (implementing the application-level model components). Developing and demonstrating such a semi-automatic prediction strategy requires extensive research and is not an explicit goal of this thesis.

6. *Portable Performance Descriptions*: Language extensions for performance properties would make the description of such properties intrinsic to the application code itself. This means that all properties would be "carried" with the application as it was built on different platforms, and particular platform-dependent performance analysis tools need not be relied upon.

In *principle*, some of the above capabilities could be obtained using a performance monitoring library instead of language extensions[1]. The fact, however, is that even the simplest of these (e.g., performance self-monitoring) are used in only very restricted ways (e.g., to report simple cumulative metrics). Runtime adaptation is the only behavior that is known to be regularly implemented in application code via the use of performance monitoring libraries or ad hoc code.

We believe that this lack of performance diagnosis code in application software is due to fundamental limitations of performance monitoring via libraries, and that these limitations would be removed by introducing language extensions for expressing performance properties. In particular, *there are 4 technical advantages (and a subjective advantage) to having language support for expressing performance properties*, as opposed to the more traditional library-based approach:

- Library calls for performance monitoring within an application can inhibit compiler optimizations in some cases, hurting the performance of the application. Language extensions could be recognized and dealt with in the compilation process to minimize or reduce the impact on optimization, as described later.

- The library approach makes fine-grained *dynamic* control of instrumentation quite difficult, if not impossible. This is due to the lack of knowledge inherent in the library approach vìs-a-vìs the program's structure and potential instrumentation sites.

- The notion of a flexible instrumentation-aware compilation system is difficult to imagine using the traditional library approach.

- The semi-automatic performance prediction described above cannot be realized with libraries.

[1]Note that external performance tools do *not* provide these capabilities; these capabilities require explicit code to be incorporated directly within the application, e.g., via a performance monitoring library.

- Finally, only language support for performance properties allows programmers to make first-class associations between performance properties and *specific* portions of the program code. The library approach forces programmers to write ancillary code to express any such associations.

1.3 Overview and Contributions of This Thesis

This thesis proposes a small but flexible set of language mechanisms that can be used to express performance properties of a program within the application source code. It also addresses two key challenges that arise in implementing these language mechanisms: *doing so without requiring support from vendor compilers* and *minimizing interference with compiler optimizations*. It is our opinion that both of these could be important to facilitate adoption of the presented ideas in practice. The specific technical contributions of the thesis are as follows:

1. A small set of language mechanisms that can be used to express and monitor arbitrary performance metrics for an application, and that defines a few widely used metrics in terms of these mechanisms. Any performance metric can be applied to an arbitrary point or an arbitrary *scope* of a program. The language mechanisms are based on C but should be straightforward to extend to other lexically scoped languages. Furthermore, we have designed our language extensions to be extremely flexible while intruding as little as possible into the target application code.

2. An "instrumentation binding" mechanism which allows for instrumentation to be dynamically added or removed from a particular program region. The manner in which instrumentation is added or removed is under the precise control of the application program itself. This binding mechanism also permits the generation and use of a production executable (i.e., where the presence of disabled instrumentation incurs negligible cost) of the application without the recompilation required to achieve the same result using traditional preprocessor mechanisms.

3. An implementation of these mechanisms that *does not require support from a vendor compiler*. In particular, the implementation is divided into multiple phases: (a) a simple source-to-source preprocessing phase that gathers information about the required performance properties and

leaves lightweight markers at instrumentation points, and (b) post-link binary editing and runtime instrumentation phases that transform the code, insert the necessary performance instrumentation, and manage metric queries.

4. A novel technique to minimize interference with compiler optimizations due to the presence of instrumentation sites. In particular, the first compiler phase inserts simple operations on `volatile` variables as markers to communicate all "location" information to the subsequent compiler phases. Such operations inhibit fewer compiler optimizations than explicit function calls to instrumentation routines. (Note that this interference would not be an issue if the vendor compiler itself supported performance annotations directly, as discussed in Chapter 4.)

5. The implementation of these techniques using the LLVM link-time compiler infrastructure [24]. We developed some stand-alone tools as well. Experiments are presented showing how the language mechanisms can be used to implement performance self-monitoring and performance assertions in POV-Ray, a well-known raytracing code, and in a long-running distributed proxy server application. Also presented are empirical results that show the runtime overhead incurred by our approach.

The broader impact of these contributions could be to change the way that programmers think about the performance aspects of their code. In current software development practices, performance analysis is of vital concern to developers, but it is essentially divorced from the development of the application itself. It is our hope that by providing robust and flexible performance instrumentation mechanisms at the language level, the performance properties of programs can be encoded in tandem with or as a fundamental component of the application logic itself.

1.4 Outline of This Thesis

The next chapter compares the approach presented in this thesis with related work in the literature. Chapter 3 presents the language methodology and semantics, and some of the predefined performance metrics. Chapter 4 describes the compiler implementation techniques, and describes the implementation in detail. Chapter 5 presents our experience with using these language mechanisms with POV-Ray, a production quality public-domain raytracing application, and in a distributed proxy server code. Finally, Chapter 6 concludes with a summary of the contributions made by this thesis.

Chapter 2

Previous Work

The notion of describing performance properties in application code is not truly novel, but many different approaches have been taken, each usually focusing on a very specific feature of performance-oriented programming, e.g. performance prediction, the collection of and reflection upon variable value histories, compiled languages that can describe instrumentation, and so forth. The purpose of this chapter is to delineate this previous work, and to distinguish it from the material presented in this thesis.

2.1 Language-Level Performance-Specific Mechanisms

The work presented in thesis can most aptly be compared against a system by Hollingsworth et al. called MDL [21] . MDL has many features in common with this thesis, summarized as follows:

- A language-driven approach that allows for the specification of metric computation code that can be placed into the application program.
- The manner in which the code is inserted can be specified, e.g. specifying what points in the application are to be instrumented.
- A method for dynamic instrumentation that permits some control over which metrics are evaluated during runtime.
- The decoupling of metrics from program components such as modules and procedures.

However, this thesis significantly improves on MDL by:

- Proposing language primitives that permit instrumentation points to be specified *at the same granularity as the application's source language*, e.g. at program statement and arbitrary language scopes[1]. Currently, MDL can only instrument at procedure entry and exit points, and individual call statements.

- Presenting more flexible dynamic control mechanisms. Our mechanisms for dynamic control are better because the dynamic insertion and deletion of instrumentation is controlled by the application logic itself.

- Proposing language extensions that can be used for performance self-monitoring. MDL provides no way for diagnosis code to be written in the application code, and thus no way for the application to use the results of metric computations.

One language feature unique to MDL is its ability to apply a particular metric to "globbed" code resources, such as "all procedures". The design and implementation of similar mechanisms is part of our future work. However, we believe that the above key features of this thesis sufficiently distinguish our work from MDL.

The idea of "program histories", proposed informally by Proebsting and Zorn [31], has several goals in common with this thesis. In fact, one of their goals is even more general, namely, to enable a program to record the history of arbitrary program values (including values of variables or values of performance metrics), and perform queries on these histories. For performance values, these ideas are very similar to the notion presented here regarding bounded and unbounded series accumulator types. Their language mechanisms for performance are not extensible, however, i.e., they do not enable user-defined metrics to be easily defined for arbitrary code constructs. Finally, they do not describe any implementation strategies and to our knowledge have not implemented their language proposal.

The idea of describing performance assertions and checking them at runtime (as demonstrated in both POV-Ray and the distributed proxy code) is not unique to this work. In particular, Vetter and Worley [38] describe a method that allows performance assertion specification via library calls, but the lexical constructs for assertion specification are parsed and evaluated dynamically, and

[1]The drawback to this is that we do require application source code, whereas MDL does not.

no attempt is made to devise true language-level mechanisms. Although performance assertions were not one of explicit goals of this thesis, the approach presented here is able to fully subsume the method that they describe. Perl [29] has also done work on performance assertions, but the performance assertions are evaluated offline and thus are different than approach we use (which evaluates performance assertions online). However, Perl's work approach does present a language for describing performance assertions and requirements.

2.2 Runtime Techniques and Libraries

Buck and Hollingsworth [7] present a novel method of performing instrumentation non-intrusively, at runtime, by inserting dynamically-generated code into active programs at predefined, fairly coarse-grained instrumentation points. However, their approach is fundamentally different than that of this thesis, because the code snippets to insert at instrumentation sites are specified externally by a client of DynInst. These snippets are created using a specialized API for constructing the C++-like AST structures. This differs significantly from the approach presented here, which seeks to provide ways to encode the performance properties into the base application. Also, the DynInst approach requires the presence of debugging information in the target executable, which likely implies that many important compiler optimizations must be disabled.

The PACE project and related work [13, 1] seeks to characterize performance aspects of parallel applications in the predictive sense, but does not employ any language-level mechanisms to do so. These works primarily seeks to unify predictive capability with the performance features of the base application.

Significant work has been done on binary rewriting to achieve post-link instrumentation [7, 23, 37, 16], some specifically focusing on performance instrumentation. However, none of these attempt to address performance instrumentation issues at the language-level. Also, most are geared towards providing flexible APIs for describing post-link transformations in order to facilitate the construction of tools which perform binary rewriting, a feature which is only tangentially related to this work.

Anderson et al. [4] present an online performance instrumentation method called "continuous profiling" which frequently samples performance data in a non-intrusive, low-overhead manner (1-

3% slowdown on average). The goal of continuous profiling is to provide instantaneous access to fine-grained profiling data even in production environments. Continuous profiling differs from the work presented in this thesis in the same way as other profiling approaches (e.g., [5, 34, 17]), in that the synthesis of application logic and empirical performance data is not one of its goals.

The PAPI project [6, 11] provides a portable interface for querying hardware counters, and strives to facilitate overall application performance monitoring using a traditional, albeit highly portable, performance instrumentation library approach. The major advantage of any library-centered approach, including theirs, is that it is relatively easy for programmers to adopt since it does not not require any compiler support. On the other hand, the approach also has some significant limitations relative to the goals of this thesis. Perhaps most importantly, it does not provide a straightforward mechanism to relate performance to application-level code constructs such as loops, statements, or functions, which we consider to play a key role in enabling the application developer to quickly identify performance bottlenecks. Second, their approach requires function calls to be inserted for performance monitoring at the source level, thus inhibiting many compiler optimizations that must cross the relevant instrumentation points. Finally, the approach can be significantly more difficult to use because extensive application code must be inserted to initialize the library and invoke the desired metrics. However, we successfully encapsulated aspects of PAPI with the language features presented in this thesis, thus greatly simplifying the use of their metrics and providing a useful, portable monitoring layer for processor performance metrics.

Chapter 3

Performance-Oriented Language Extensions

Fundamentally, the language extensions presented here provide mechanisms that the programmer can use to describe values in the target code that need to be monitored or collected, as well as what statistical results need to be obtained for the collected metric datasets. In addition to providing built-in metrics, the language extensions permit the programmer to define arbitrary metrics by providing their own. These user-defined metrics can then be employed in exactly the same way as the built-in metrics. The language extensions also provide support for many kinds of common statistical operations on data sets: standard deviation, variance, averages, moving averages, etc. The programmer may also provide custom code that operates on the collected data using standardized interfaces.

3.1 Overview of Language Extensions

Consider the example code shown in Figure 3.1. It is referred to throughout the following sections. The example given in the figure demonstrates the use of some of the language constructs in high-level code for a simple server application. The first 3 lines of the example specify which metrics may be sampled in the application, how those samples are to be gathered into datasets, as well as the names by which the program shall refer to the metrics. As an example, consider the metric description for `elapsedHist` on line 2. The `pp_interval` construct defines `elapsedHist` as a metric that must be applied to intervals (explained below), while the `bounded_series` component of the statement denotes that the metric data will be aggregated into a fixed-width array. The size of

```
(1) pp_point<sumcount, getNumActive> activeAcc;
(2) pp_interval<bounded_series, elapsed_time_start,
        elapsed_time_end, size=100> elapsedHist;
(3) pp_interval<sumcount, l1cache_miss_start,l1cache_miss_end> cacheMissAcc;

void main_server_loop() {
        for(;;) {
            wait_for_client_request();

            /* Record number of active connections*/
(4)         declarePoint p1(activeAcc);

            /* Record elapsed time and L1 cache misses
               in the following enclosed scope */
            {
(5)         declareInterval i1(elapsedHist, cacheMissAcc);
(6)         handle_client_request();
            }
        }
}

/* Report the recorded metric values */
void func() {
        printf("Avg # connections on client request:");
(7)     printf("%d\n", pp_avg(activeAcc));
(8)     printf("Stddev of elapsed time:  %f\n",pp_stddev(elapsedHist));
(9)     printf("Avg L1 cache misses for region:  %f\n",pp_avg(cacheMissAcc));
}
```

Figure 3.1: Example code showing language extensions

the fixed array (100 in this case) is specified as a parameter at the end of the directive. The other two arguments, `elapsed_time_start` and `elapsed_time_end`, are the names of the metrics functions that are to be invoked whenever the `elapsedHist` metric is used, at the start and end of the interval, respectively.

Looking at the rest of the code in Figure 3.1, we see the definition of a valid instrumentation point at line 4, and at line 5 the definition of a valid instrumentation interval (the interval is over the lexical scope that encloses the `declareInterval` statement). The names of metrics given in parenthesis after the `declare` statements in lines 4 and 5 specify the default metrics to associate with those sites. Thus, the `activeAcc` metric will be sampled at line 4, and both the `elapsedHist` and `cacheMissAcc` metrics will be sampled over the interval defined by the enclosing scope of line 5. The samples of the metrics are queried in lines 7-11, by simply using the metric name as a program variable (usually passing it to a function to compute a particular statistic over the dataset).

```
(1)  metric-decl      ::=  metric-point | metric-interval
(2)  metric-point     ::=  pp_point<accum-type [,func [,opts]]> mname
(3)  metric-interval  ::=  pp_interval<accum-type, func, func [,opts]> mname
(4)  opts             ::=  property=value [,opts]
(5)  accum-type       ::=  scalar | count | sumcount | bounded_series |
                           unbounded_series | ...
(6)  sample-stmt      ::=  sample_begin(mname, arg1, ...)  |
                           sample_end(mname, arg1, ...)
(7)  decl-point-stmt  ::=  declarePoint ipname[(mname, ...)]
(8)  decl-int-stmt    ::=  declareInterval ipname[(mname, ...)]
(9)  bindmet-stmt     ::=  bindMetric mname to ipname
(10) unbindmet-stmt   ::=  unbindMetric mname from ipname
(11) property         ::=  size | maxSize | operator ...
(12) value            ::=  string | number | ...
(13) mname            ::=  [valid variable identifier]
(14) ipname           ::=  [valid variable identifier]
(15) func             ::=  [valid function identifier]
```

Figure 3.2: Grammar for proposed language extensions

As seen in the example, the proposed language extensions have syntax similar to existing languages such as C, and may be used to extend languages such as C, C++, and Fortran. The language syntax is shown in Figure 3.2. There are four primary language components: *metric declarations*, *metric measurement sites*, *metric binding sites*, and *metric uses*. For example, in Figure 3.1, lines 1–3 are metric declarations, lines 4 and 5 both double as metric measurement and metric binding sites, and the statements at lines 7–9 demonstrate metric uses.

One of our most important language design goals is to decouple measurement sites from the particular metric(s) evaluated there. This decoupling permits multiple metrics to be associated with one specific measurement site, single metrics to be associated with multiple sites, and sites that have no metrics associated with them at all. To achieve this goal, metrics are explicitly bound (either statically or dynamically) to measurement sites, a process called *metric binding*.

There are two types of metrics distinguished by how metric values are bound and subsequently sampled: *point metrics* are bound to and sampled at a single point, while *interval metrics* are bound to a specified interval and sampled at the beginning and end of it, and the two values are combined (e.g., by subtraction). In Figure 3.1 again, `activeAcc` is a point metric that is statically bound and sampled at line 4. Both `elapsedHist` and `cacheMissAcc` are interval metrics statically bound at line 5, and sampled at entry and exit of the innermost scope enclosing line 5 (the innermost scope implicitly defines the start and end points for sampling an interval metric). Note that only static metric binding is shown in Figure 3.1. These design choices are motivated and explained in detail below.

3.2 Sampling Points and Intervals

The `declarePoint` and `declareInterval` directives allow the user to denote *sampling points* and *sampling intervals* in the code where instrumentation may be applied. Sampling points are single locations (i.e., the exact location of the `declarePoint` directive) in the code, and metrics that are associated with points are called point metrics. Sampling intervals are defined by a *start point* and an *end point*, and metrics that are associated with intervals are called interval metrics. The start and end points of an interval are implicitly defined by the enclosing scope where a `declareInterval` directive occurs[1]. For example, in Figure 3.1, the interval declared on line 5 has its start point immediately before line 5 and its end point immediately after line 6. By using the scoping features of the language like this, we provide the user with language-level control of intervals without forcing "start" and "end" constructs to appear throughout the code, nor do statement labels have to be used, which can introduce extraneous names into the name-space of the program. This technique works uniformly for both structured and unstructured control flow.

After points or intervals have been declared, the appropriate point or interval metrics may be associated with them (i.e., metrics may be bound to the relevant instrumentation site(s)). A more detailed description of point and interval metrics can be found in Section 3.5, and metric binding semantics are described in Section 3.6.

[1]Except in the less common case where actual parameters must be provided to the metric functions. In this case, the actuals are expressed by using the `sample_begin` and `sample_end` directives (see Figure 3.2) to declare the endpoints of the intervals, with the caveat that metrics cannot be dynamically bound to intervals declared in this manner.

3.3 Metric Declarations and Variables

A metric declaration (as seen on lines 1-3 in the example) introduces a *metric variable* that can be referred to by name in different parts of the program. Metric variables are always global in scope. Separating the declaration of a metric from the location where its value is sampled allows the same metric variable to be sampled (i.e., measured) in several parts of the code, e.g., to accumulate the execution time for different pieces of a computation into a single metric.

A metric declaration specifies a metric name, a metric accumulator type which specifies how samples are to be accumulated, a measurement function or functions, and any options specific to the selected metric accumulator type. For example, the declaration on line 3 of the example provides the option `size=100`, which is an option specific to the `bounded_series` accumulator type, and informs the compilation system to create a 100-element static array to hold metric samples.

The type of the data structure created for a particular metric variable is determined by the return type of its measurement function together with its accumulator type. For example, the metric variable declared on line 2 of the example code would be of type `long*`, and it would point to the base of the array statically allocated to hold the results of up to 100 elapsed time interval samples, which are of type `long`. Uses of the metric variables occur just like normal language variables. Thus, one significant advantage of this approach is that no cumbersome language or library mechanisms must be used to manipulate the gathered metric data. Rather, manipulation of the metric datasets occurs as if the metric variables were in fact scalar or array variables in the source language.

3.4 Metric Accumulator Types

A metric declaration has two key parameters: *metric accumulator types* and *metric functions*. Metric accumulator types describe how the collected sample data are to be stored (e.g., in a fixed-length vector, in a scalar variable, etc.), and metric functions are invoked to sample values for a metric. Small metric functions can be implemented as preprocessor macros for efficiency[2].

[2]When the metric is statically bound to a point or interval. The preprocessing phase must be able to expand the macro and place the code at the particular instrumentation site(s), after which point the metric cannot be unbound.

There are only five fundamental accumulator types: `scalar`, `count`, `sumcount`, `bounded_series` and `unbounded_series`. In the example, the metrics `activeAcc` (declared on line 1) and `elapsedHist` (declared on line 2) have accumulator types `sumcount` and `bounded_series`, respectively.

A `scalar` is just like a regular program value, and holds one sample value at any given time. A `count` metric represents an increment of a counter at a particular point in the program. A `sumcount` metric implicitly maintains a pair of double-precision variables that represent an instance count (i.e., how many times the metric has been sampled) and a summed value (i.e., the sum of all sampled values). Note that this provides a lightweight and efficient mechanism for computing the average of a large series of sample values, as long as no other statistics are required. The `bounded_series` metric accumulator type is simply a finite vector of sample values that implements an ordinary moving window of sample values. The `unbounded_series` accumulator allows an unpredictable number of samples to be accumulated, using a dynamically growing vector. The runtime system raises an out-of-memory exception if the vector grows too large[3].

Included as part of the standard metric library are a number of common statistical functions (average, standard deviation, variance, etc.) that operate on metric variables. The functions can be invoked in the user code as normal function calls, and the metric variable names are passed as parameter(s), e.g., the call to `pp_stddev` and `pp_avg` on lines 8 and 9 of the example in Figure 3.1. Of course, the user may define custom statistical functions and use them in place of those provided, which allows them to express complex performance data analysis in the source language of the application itself.

3.5 Point and Interval Metrics

As noted earlier, metrics can be sampled in two distinct ways with respect to the execution of the target program: at a single point or at the beginning and end of an interval. A point metric variable is declared using `pp_point`, e.g., `activeAcc` on line 1 of Figure 3.1. It specifies a single measurement function used to compute the metric value. Points in the program source that can be instrumented are identified by the user, as described in Section 3.2. A metric value may be sampled at any such instrumentation point by explicitly binding the metric to the point (see Section 3.6).

[3]The current implementation does not support unbounded series metric types.

An interval metric variable is declared using the `pp_interval` construct, and specifies two measurement functions to be applied at the interval endpoints. For example, the elapsed time of an interval can be measured using the functions `elapsed_time_start` and `elapsed_time_end`. As with point metrics, interval metrics may be bound to an arbitrary number of intervals. An interval metric is sampled as follows. When execution reaches an interval to which the metric has been bound, the metric's first measurement function is invoked, and its value is saved to a temporary. This temporary is then passed as a parameter to the metric's second measurement function, which is invoked when execution reaches the end of the same interval (or any function exit points that occur in the interval). This latter measurement function samples the second value, computes the difference (or any other combination) of the two values, and returns the result. *This* result is the metric sample value for the given interval.

As an example, consider the sample interval shown in Figure 3.3, where the metrics are defined the same way as in Figure 3.1.

```
        {
(1)        declareInterval i2(elapsedHist, cacheMissAcc);
(2)        if(...)
(3)           goto someLabel;
(4)        handle_client_request();
        }
```

Figure 3.3: Multiple exit points from an instrumentation interval

Here the second measurement function for each bound interval metric (i.e., `elapsed_time_end` and `l1cache_miss_end`, respectively) is invoked both before the `goto` and before the closing '}' for the scope.

3.6 Metric Binding Semantics

As described in Section 3.5, a metric may be may be sampled at declared points or over declared intervals by explicitly binding the metric to the point or interval. Point metrics can be bound to sampling points (and, likewise, interval metrics to sampling intervals) either statically or dynamically. For example, in Figure 3.1, the static binding of the point metric `activeAcc` to the sampling point `p1` is accomplished by naming the `activeAcc` metric in the list of initial bindings for `p1` (see line 4). This means that whenever execution reaches the instrumentation point `p1`, the `activeAcc` metric is computed and any relevant data structures are updated. The same binding mechanism is used to bind interval metrics to sampling intervals (e.g., in line 6 of Figure 3.1, the interval metrics `elapsedHist` and `cacheMissAcc` are statically bound to interval `i1`).

Figure 3.1 demonstrates only static binding of metrics to points or intervals. The precompilation phase recognizes these static bindings and generates the appropriate code to ensure that the metric(s) will be computed the each time execution reaches the points or intervals.

Dynamic binding occurs when a new association is between a particular declared metric and a particular sampling interval or point occurs at runtime instead of at compile-time. The `bindMetric` directive is used to accomplish this (see Figure 3.4).

```
        {
(1)         declareInterval i3;
(2)         if(...)
(3)             handle_client_request();
        }
(4)     bindMetric cacheMissAcc to i3;
```

Figure 3.4: Dynamic binding to an instrumentation interval

In the figure, there is no static binding specified for interval `i3`, indicating that no metrics are initially associated with the interval. Rather, when the `bindMetric` directive is executed the first time, the `cacheMissAcc` metric is added to the set of metrics that are bound to the interval. The *next* time that interval `i3` is executed, the L1 cache misses of the region will be computed and aggregated in the manner specified by `cacheMissAcc`'s accumulator type[4]. In a similar manner, the `unbindMetric` directive can be used to remove the association between a particular metric and instrumentation site[5]. The ability to dynamically bind and unbind metrics to and from instrumentation sites is one of the key features of the system.

Our approach, then is to decouple metric specifications from the instrumentation sites where

[4]There is nothing to prevent the `bindMetric` directive from appearing inside the interval to which it applies. However, any new metrics will not be computed until the next time the interval is executed.

[5]Metric unbinding has not implemented for this thesis, but the design for its implementation exists and is straightforward.

metrics are evaluated. Without decoupling, metric specification necessarily occurs at the instrumentation sites, which is only practical for the most simple instrumentation [21]. Thus, decoupling has two primary advantages. First, the decoupling permits instrumentation to be specified in a straightforward manner; metrics are described in the abstract and then applied to particular sites. Secondly, decoupling makes dynamic binding possible. If metrics had to be specified at instrumentation sites, any binding is a static property of the program, and new instrumentation could not be added to or removed from the sites. Since dynamic control of instrumentation and straightfoward metric specification are valuable tools, we use the decoupled approach in this work.

3.7 Metric Functions and Directives

Metric functions are the functions invoked by the runtime system when sampling a particular metric. A metric function can be "built-in", in which case it comes from a standard library of metrics provided as part of the compilation system, or it can be a custom function provided by the user. The example (Figure 3.1) illustrates both built-in metric functions and a user-defined metric function. The `elapsedHist` metric uses the `elapsed_time_start` and `elapsed_time_end` metric functions (which are called at the start and end of interval `i1`, respectively), which come from the standard metric library. Likewise, the `cacheMissAcc` metric uses the functions `l1cache_miss_start` and `l1cache_miss_end`, also from the standard metric library. The metric `activeAcc` uses the user-defined function `getNumActive`, effectively specifying that is to be called whenever a new sample is required for the `activeAcc` metric. Functions of any type may be used, but they must return a primitive scalar type if the built-in statistical operations are to be used on them (otherwise, user-defined statistic functions must be provided).

For interval metrics, the user specifies both a “begin interval” function and an “end interval” function, as described above. The former requires no special arguments, while the latter does: one (implicit) formal parameter, which is a pointer to the return value of the corresponding start-site metric function. This is to permit automated passing of data between the two functions. See Section 3.5 for more information on interval semantics.

Using the proposed language extensions, the user has a great deal of flexibility when describing the locations in the code where a metric is sampled, and what parameters are passed to the metric function(s). When a metric function for a certain metric takes no parameters, which is the most common case, simple sampling directives (i.e., `declarePoint` or `declareInterval`) are sufficient. However, if the user wishes to use metric functions with formal parameters, the language extensions provide an alternate version of the directives (see Figure 3.2) that allows actual parameters to be specified, with the caveat that dynamic binding of such metrics to sampling points or intervals is not possible[6]. Otherwise, the semantics of the these alternate directives are identical to their 0-argument counterparts.

[6]The expressions computed and passed to the metric functions are context-dependent on the scope of the sampling point or interval, and so the dynamic binding mechanism used for 0-argument metric binding cannot be used.

Chapter 4

Compiler and Runtime System

One of the primary goals of our compiler implementation strategy is to avoid requiring support from vendor compilers. This goal could be important in facilitating the adoption of a performance programming methodology such as the one proposed by this thesis. It is also important in order to achieve portability (of the performance-oriented code) across compilers. For the implementation solution presented here, no compiler support is required, because all of the proposed mechanisms are realized using features of the source language itself, plus some runtime transmogrification of binary code.

Unfortunately, this goal of vendor independence entails a major technical difficulty. Since the vendor compiler performs arbitrary transformations on the input code, the locations of the precompilation instrumentation sites are lost, and must be somehow rediscovered. Furthermore, data must be maintained that tracks *which* metrics are statically bound to instrumentation sites. There is *no* reliable technique to record particular locations in the code when the compiler is treated as a black box. Naïve solutions to this problem (e.g., using opaque function calls to mark instrumentation sites) are not acceptable because they can inhibit compiler optimizations, thus increasing the performance perturbation due to the presence of performance instrumentation.

Performance perturbation due to instrumentation occurs in 2 broad ways: (i) inhibiting compiler optimizations due to the presence of monitoring code (e.g., functions calls); and (ii) affecting runtime performance itself, including increased runtime overhead, different cache behavior, different memory usage patterns, and many more. The latter is unavoidable when code is added to the program and can only be mitigated via the use of lightweight monitoring routines (which are orthogonal to this work). The former, however, can be reduced or eliminated via our implementation strategy, in some cases.

In order to solve these problems and sufficiently attain both competing goals of vendor independence and minimization of perturbation, the compiler strategy is to split actions performed on the application code into multiple phases: a precompilation phase, and multiple runtime phases. Each phase of the compilation strategy is described in turn below.

4.1 The Precompilation Phase: Marking Instrumentation Sites

The precompilation phase of the compiler (referred to as phase 1) has the following responsibilities:

- Parsing all performance language extensions used in the target code.
- Inserting special markers in the code that denote instrumentation points or intervals.
- Storing sufficient information in a persistent manner so that the runtime phases can make their transformation(s) to the code.

Thus, the overall strategy employed by phase 1 is to make minimal code changes and to gather and save relevant data.

Each metric measurement corresponds to one or two calls to a metric function or functions. Inserting these function calls directly in phase 1 (and keeping the metric function implementations hidden from the vendor compiler to prevent their inlining) would ensure that the instrumentation locations are preserved. This approach, however, could inhibit many code motion and code reordering transformations that are fundamental to modern compiler optimizations, in particular, any that need to move non-trivial computations across the instrumentation point (see below for an example). What is needed are lightweight markers that interfere with fewer optimizations, and yet are not themselves reordered or eliminated by any vendor compiler. We will replace our language directives that denote instrumentation sites with these lightweight markers (e.g., markers would be placed near lines 3 and 12 in Figure 4.1).

Note that this would *not be a problem* if access is provided to (and potentially significant modifications made to) the optimizing compiler itself, in order to ensure correct handling of the designated markers. For example, "performance annotations" could be implemented – tagging

```
int main(int argc, char** argv) {
    int i, j, val;
    {
        declareInterval i1;
        for(i = 0; i < 1000000; ++i) {
            /* val = opaqueFunction(); */
            for(j = 0; j < 5; ++j) {
                a = argc;
                b = a;
            }
        }
    }
    return a + b;
}
```

Figure 4.1: Code before the precompilation phase

special variables and using reads of those variables as markers so that the compiler preserves the locations of those reads. The drawback, of course, is that this does not achieve the primary goal of compiler independence.

This thesis proposes a novel solution for languages like C, C++, and Fortran[1] that inhibits some compiler optimizations but not many others: using operations on volatile variables as markers. The relevant property of volatile variables is that operations on them are considered to have side-effects. In particular, they must be read (and written to) the same number of times and in the same order, both before and after program optimization. For example, if a read of a volatile variable is in a loop with a constant bound, and the optimizing compiler unrolls the loop a certain number of times, the read of the volatile variable is placed appropriately within each unrolled loop instance. Similarly, the compiler cannot hoist the global volatile read out of loops because it must execute as many read operations as the original code. This is *exactly* what is meant by a marker being "carried" by whatever optimizations are performed. Thus, a mechanism exists that can be used as the special marker: a read of a designated global volatile variable is placed at each instrumentation site (see Figure 4.2).

Note that it is acceptable for some code to be moved out of an interval being measured, assuming that the same transformation also occurred in the original application. For example, if an interval

[1]Or any language that has a C-like semantics for volatile program variables.

```
volatile short gv1, gv2;
char tmp1, tmp2;
int main(int argc, char** argv) {
    int i, j;
    tmp1 = (char) gv1;
    for(i = 0; i < 1000000; ++i) {
        for(j = 0; j < 5; ++j) {
            a = argc;
            b = a;
        }
    }
    tmp2 = (char) gv2;
    return a + b;
}
```

Figure 4.2: Code after the precompilation phase, with markers at instrumentation sites

directive is inserted that is used to time the body of a loop iteration and some code is hoisted out of the loop (in both the original and the instrumented code), then the sampled values of the loop body in the original and instrumented code will be comparable, because the presence of the global volatile markers should never inhibit the code motion. To see this, refer to Figure 4.2; the reads of `gv1` and `gv2` are the markers that delimit the sampling interval. In the original code (Figure 4.1), if the vendor compiler hoists the assignments in the inner loop to outside both loops, then it is *not* prevented from doing the same transformation when the markers are present.

No assumptions are made regarding the action of the optimizing compiler, other than the fact that it handles volatile variables properly. Even if the compiler handles global volatile reads *very* conservatively, the resulting optimized code is likely to be no worse than the version with opaque function calls. However, the compiler will often be able to do much better than that. In particular, volatility semantics permit the compiler to *move arbitrary computations on non-volatile data across a reference to a volatile variable.* For example, optimizations such as dead-code elimination, constant propagation, value numbering, common subexpression elimination, loop-invariant code motion, partial redundancy elimination (PRE), strength reduction, loop strip-mining, *local* instruction scheduling (in most cases), global register allocation, and peephole optimizations should not be inhibited by references to volatile variables [28] (note that *all* of these could be inhibited, to some extent, by opaque function calls). Optimizations that *reorder* loop iterations, e.g., loop interchange, loop distribution, loop fusion, loop tiling, unroll-and-jam, or software pipelining would be inhibited, however, because they would alter the order of execution of the original references [28]. The only way we know of to prevent interference with such optimizations is to avoid placing instrumentation sites inside loops.

We tested the effectiveness of using volatile variables as markers with a simple example, using the GNU C compiler for Sparc and the Sun Workshop C compiler (both with optimization turned on). The code in Figure 4.1 is compiled in two ways: (i) with an opaque function call in the body of the outer loop (i.e., uncommenting line 6 in the figure), and (ii) with a global volatile marker in place of line 6 in the figure. The result is that the loop-invariant statements (lines 8-9) are indeed moved to the outer loop preheader when using volatile variables, but not when the function call occurs inside the loop. This illustrates how the approach presented here enables common scalar optimizations such as LICM and GCSE that are prevented in the presence of an opaque function call.

The final step is to record information that must be communicated from the precompilation phase to the runtime phases. In particular, the later phases must have some way of determining what the metric variables are, finding the reads of global volatile variables that correspond to instrumentation sites, and knowing what code to put in place of the global volatile reads. To this end, the first phase creates and initializes a static, global data structure called the global metric information table (GMIT). The GMIT contains the following information:

- The accumulator types of all metric variables.
- Any optional metric parameters (e.g., the value of the optional parameter `size` for metrics with accumulator type `bounded_series`).
- Pointers to any measurement functions used in all metric specifications, and the mapping between each metric variable and the appropriate measurement function(s).
- The type of all sampling points[2]. Valid types are "point", "interval start", or "interval end".
- The addresses of all global volatile marker variables, and the mapping between sampling

[2]A sampling interval is described by two distinct sampling points.

points and these addresses. This mapping describes which markers correspond to what instrumentation sites.

- The lexical names of all points and intervals.
- Any static bindings between metrics and sampling points or intervals.

Note that there is no attempt made to track the locations of particular sites, which are free to be moved about as a result of compiler transformations. The final locations of instrumentation sites are discovered by the runtime phases, as described in Section 4.2.

4.1.1 Implementation Details

We implemented the precompilation phase using the LLVM compiler infrastructure [24], in the form of an LLVM pass. The existing LLVM frontend for C is used, unmodified, for convenience; writing a custom frontend for the language extensions is avoided. In order to the recognize the language extensions using an unmodified C frontend, a "significant functions" approach is used. This is a common implementation technique that allows existing frontends to be extended transparently. For example, to realize the semantics of a directive such as `declareInterval`, special functions `pp_declareIntervalStart` and `pp_declareIntervalEnd` are used to denote interval endpoints. These special functions are searched for by the precompilation phase. All matching pairs[3] of calls to these significant functions determine the locations in the code that need to be transformed by the precompilation phase.

A similar mechanism is used for all of the language primitives. All lexical parameters to these primitives are given as string arguments to the representative significant functions, and these strings are parsed during the pass. Note that this is merely another implementation convenience.

After all source-level instrumentation sites are gathered, each is replaced by the read of a different global volatile variable. The implementation actually places one assignment statement which describes two operations: a read of a global volatile variable of type `short`, and a write to a temporary variable of type `char` (see Figure 4.2). These statements will be compiled into a "load half-word, store byte" instruction sequence. This particular sequence is used to reduce the number

[3] A match is determined by finding the only two uses of a "connector" scalar operand. Dataflow analysis is not needed because of the SSA intermediate representation employed by LLVM.

of potential instrumentation sites discovered at runtime by phase 3 (see Section 4.2.3 for details). Additionally, when a site is processed, its GMIT entry is constructed. The GMIT is implemented as a static, global `struct`. The addresses of the volatile variable markers contained in the GMIT are resolved by the linker. GMIT contents are discussed in the previous section.

The output of the precompilation phase is generated from the LLVM C backend, and can then be compiled normally using a vendor C compiler.

Finally, please note that the our use of LLVM is orthogonal to the rest of the work; any C-to-C compiler could have been used, or a custom frontend could have been written that performed the described source-to-source transformations.

4.2 The Runtime Phases

The broad goal of the runtime phases is to locate the markers that denote instrumentation sites, and to stitch in the appropriate metric function invocations at these sites using the mechanisms described in Section 4.2.1. However, locating the the instrumentation sites (which are now denoted by reads of volatile global variables) is difficult for a number of reasons. The most significant difficulties are as follows:

- Finding a read of a volatile variable is hard, even though its address is already known[4]. Two reasons for this difficulty are: (i) constant values can be obscured by the sequence of machine instructions[5] used to load the address into a register, and (ii) certain optimization techniques (e.g., value numbering) can generate instruction sequences that load constant addresses into registers in myriad ways and later reuse these values, making it impossible to search for fixed instruction sequences.

- Common compiler optimizations such as function inlining can further obscure the locations of marker sites, because it is not known which particular post-compilation functions are instrumented – inlining may have effectively removed the instrumented functions.

[4]Via the GMIT constructed by phase 1.

[5]These sequences are particularly long on 64-bit RISC architectures, because so many instructions are required to load a 64-bit constant value into a register. Extracting constant values from such sequences is difficult, but not impossible.

```
func:
    inst1
    inst2
    ...
    lduh [%o0], %o1
    ...
    lduh [%o3], %o0
    ...
```

Figure 4.3: Candidate function with undiscovered instrumentation sites

Runtime Phase	Responsibilities
Phase 2	Iterate over each user-code function, write a branch to a phase 3 trampoline at each entry, and write contents of those trampolines
Phase 3	In each invoked user-code function, locate its load candidates, write a branch at each candidate site to a phase 4 trampoline, and write contents of those trampolines
Phase 4	Finalize the set of valid instrumentation sites, write a branch at each to an instrumentation trampoline, and write contents of those trampolines
Execution Monitoring	Invoke metric computation for all metrics bound to a particular instrumentation site

Table 4.1: Runtime phases and their responsibilities

To solve these issues effectively and efficiently, our implementation uses a low-overhead, multi-phase runtime approach that, via a kind of iterative refinement across phases, rediscovers the precise locations of the instrumentation points in the code. Each runtime phase will "zero in" on the *actual* instrumentation sites. The transformation phases only run "once" (they execute once for each instrumentation function or instrumentation site), and only for functions and sites that are reached during execution. Thus, their cost is amortized across the execution of the program.

The runtime phases begin by locating load instructions that are likely to be the instrumentation markers; these are *load candidates*, and a function which may contain instrumentation sites is called a *candidate function*. Figure 4.3 demonstrates such a function in Sparc assembly pseudocode.

There are three distinct runtime phases, phases 2-4, respectively. Table 4.1 describes each phase and its responsibilities.

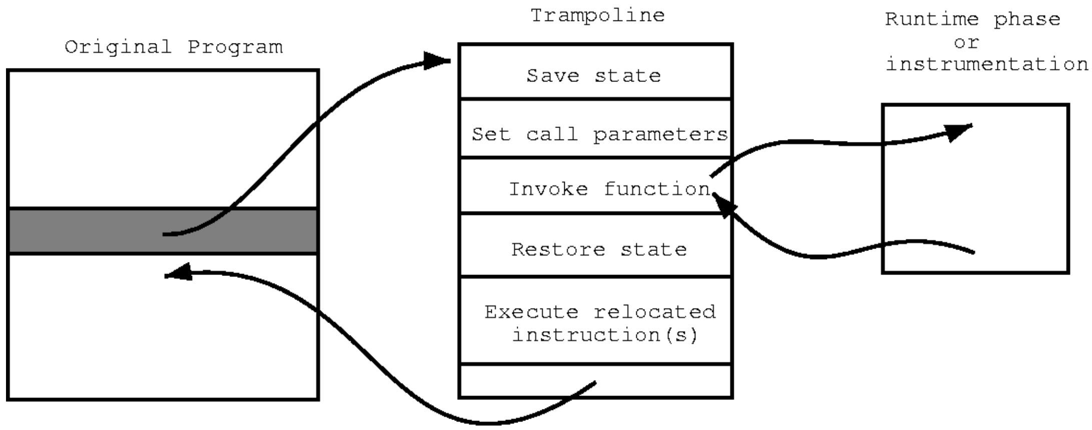

Figure 4.4: Using trampolines to invoke arbitrary functions

4.2.1 Trampolines and Branching Mechanisms

At runtime, a mechanism is needed that permits insertion of code that invokes either later runtime phases or the instrumentation itself. One way that this is commonly achieved (e.g., in [7] and [21]) is via an implementation mechanism known as a *trampoline*. A trampoline is a special code region that is reached during execution after having overwritten original program instructions with branches (see Figure 4.4). As shown in the figure, this code region handles all saving and restoring of execution state, sets up parameters for any function(s) that it calls, and invokes those function(s). The overwritten instructions are executed after the execution state has been restored, and control is returned to the original code. Instead of invoking function(s), the "invoke function" step in Figure 4.4 can be replaced with other code that jumps to another region of code or trampoline. This means that an arbitrary number of trampolines can be chained together if desired.

Trampolines are used in phases 2-4 to perform particular phase-specific transformations (including the construction of trampolines for later phases), and during execution to control the invocation of instrumentation function(s) at instrumentation sites. Throughout the following sections, the phrase "phase n trampoline" is used to refer to a trampoline that is responsible for invoking runtime phase n on the program.

```
func:
    branch to phase 3 trampoline
    inst2
    ...
    lduh [%o0], %o1
    ...
    lduh [%o3], %o0
    ...
```

Figure 4.5: Candidate function after phase 2

4.2.2 Phase 2: A Lightweight Program-Wide Transformation

Phase 2 executes exactly once at program startup. Its broad goal is to insert a branch to a phase 3 trampoline at the start of each user-code function[6]. Placement of these branches is restricted to user-code functions so that instrumentation sites are not searched for where they cannot occur. To find the user-code functions, symbol table information is read from the ELF executable, which contains the start addresses and sizes of the functions. For more details regarding some issues with finding specific functions in a compiled binary, see [23].

For each user-code function, phase 2 overwrites one or more instructions at the function entry. Thus, there is a one-time cost at program startup for making program-wide transformations that will later invoke other phases. The experimental results in Section 5.1 show that this cost is small; even for large programs, phase 2 is not cost-prohibitive, because examining the ELF data is quite fast. One of the major benefits to this approach is that only extremely light work is performed for those functions that are never invoked, since the later runtime phases are never invoked for such functions.

Figure 4.5 shows the code from Figure 4.3 after phase 2 has been applied to it.

4.2.2.1 Implementation Details

As previously noted, phase 2 applies its transformation only to user-code functions. Failing to do this creates an interesting problem that warrants explanation. Consider what happens if phase

[6]User-code functions, in this context, are considered to be functions that do not come from the runtime system or from system libraries. In our implementation, these are any functions not excluded by the exclusion utility (see Section 4.2.2.1).

2 does not discriminate in this manner, and is instead applied to every function discovered from the ELF executable's symbol table (in fact, an early implementation did precisely this). The following scenario can occur: let us assume that the first candidate function that is invoked at runtime is a user-code function. When it is invoked, the `phase3` function (which effects the phase 3 transformation) is called from the phase 3 trampoline. Now, let us also assume that the `phase3` function calls a standard library function, `libFunc`, which has also been transformed by phase 2. As described in Section 4.2.3, any branch instructions that transfer control from a candidate function to a phase 3 trampoline are removed by phase 3. However, `libFunc` can be called from `phase3` before this actually occurs. When `libFunc` is called, it immediately branches to a phase 3 trampoline (because it has been transformed by phase 2), which will eventually call `libFunc` again. However, the branch to at the entry to `libFunc` has not yet been removed and so these steps are repeated indefinitely (see Figure 4.6). Similar problems may occur for any function processed by phase 2 that is call-graph-reachable from the functions that implement the runtime phases.

To solve this problem, a command-line utility called `mkexcl` ("make exclusions") was designed, implemented, and integrated into the build process. Before the final executable is linked, `mkexcl` is invoked on all libraries and object files whose symbols should be excluded from consideration by phase 2: all libraries that compose the runtime system described in this thesis, and all relevant system libraries. The output of `mkexcl` is C++ code that can be invoked at runtime to build an STL set containing the excluded symbol names; this code is compiled and linked into the final executable. When phase 2 executes, it first invokes the `mkexcl`-generated function to populate the exclusion set and then checks each function symbol for membership in the set before processing it. This solution is quite unobtrusive[7] and provides an effective solution to the problem. Furthermore, if large pieces of user code are known not to contain instrumentation sites, `mkexcl` provides the user with a clean way to specify this information.

4.2.3 Phase 3: Finding Potential Instrumentation Sites

Phase 3 executes once for each function that is invoked at runtime. It is responsible for finding load candidates in these candidate functions. That is, for a given user-code function, phase 3 constructs

[7] Because our implementation already requires a modified linking step, requiring the invocation of a small, fast command line utility does not seem overly burdensome.

User-code candidate function
↪ Phase 3 trampoline → phase 3 function
  ↪ Library function transformed by phase 2
    ↪ Phase 3 trampoline → phase 3 function
      ↪ Library function transformed by phase 2
        ↪ ...(last 3 steps repeat forever)
    ↪ **Not reached:** Remove branch at library function entry point

Figure 4.6: Infinite recursion in a naïve phase 2 implementation

a superset of the actual instrumentation sites contained within that function. Since phase 3 is only executed once per invoked candidate function, its cost is amortized across the execution of the program.

```
func:
    inst1
    inst2
    ...
    branch to phase 4 trampoline
    ...
    branch to phase 4 trampoline
    ...
```

Figure 4.7: Candidate function after phase 3

The first task that phase 3 performs is the restoration of the original instruction(s) (overwritten by phase 2) to their original locations in the candidate function. After this, phase 3 scans the instructions of the candidate function body and heuristically locates load candidates, i.e., instructions that look like they may be a load of a global volatile variable that represents an instrumentation site. However, it can not be determined if a particular load candidate is an actual instrumentation site until execution has reached it, because the address being dereferenced is not known until then. Thus, at each load candidate in the candidate function body, a branch is overwritten to a phase 4 trampoline, and when phase 4 is invoked for a particular candidate, it determines whether or not the load candidate is in the set of actual instrumentation sites for that candidate function. After this, phase 3 returns and the program continues execution.

Because the set of actual instrumentation sites is not finalized until phase 4, phase 3 can erroneously select load instructions as load candidates. These false positives incur processing overhead in phase 4. Our implementation significantly alleviates this problem by searching for very unusual load sequences[8] that operate on the same register with no intervening writes to that register. This approach greatly reduces the number of discovered load candidates overall.

Figure 4.7 shows the code from Figure 4.5 after phase 3 has been applied to it. Note that both `lduh` instructions have been selected as load candidates, and overwritten with branches to phase 4 trampolines, where their membership in the set of actual instrumentation sites will be resolved.

4.2.3.1 Implementation Details

```
IsSparcLoadCandidate(i, endAddr)

d ← i's destination register
s ← first store instruction in range (Address(i), endAddr] with source operand d

if s has opcode stb   ▷ Detected schema 1
    return true

if s has opcode sth and is FP-relative   ▷ Suspect schema 2
    f ← FPOffset(s)
    l ← first FP-relative load instruction in (Address(s), endAddr] with FP offset f
    if l not found
        return false
    d′ ← l's destination register
    s′ ← first store instruction in range (Address(l), endAddr] with source operand d′
    if s′ has opcode sth   ▷ Detected schema 2
        return true

return false
```

Figure 4.8: Heuristic for finding load candidates on Sparc

The heuristic used by phase 3 for load candidate detection merits further discussion. Recall from Section 4.1 that phase 1 marks instrumentation sites with an assignment statement. On the Sparc architecture, this assignment statement is usually compiled (by the vendor compiler) into two instructions: a load half-word followed by a store byte. However, because the high-level assignment statement is not atomic, the compiler is free to schedule these two instructions in many different ways as long as the relevant dependences are maintained; other instructions are likely to be

[8] On Sparc, for example, a load half-word followed by a store-byte is used, as explained in Section 4.1.1.

scheduled between the load half-word and store byte instructions. Thus, the following instruction sequence is likely to occur (this sequence is referred to as "schema 1"):

```
lduh [mem1], %r[d]
...
stb %r[d], [mem2]
```

Furthermore, if the vendor compiler does not have optimization enabled or is prevented from promoting the `char` variable (the *lvalue* of the assignment) to a register, the generated instruction sequence will copy the value to the stack before the final store byte (this sequence is referred to as "schema 2"):

```
lduh [mem1], %r[d]
...
sth %r[d], [stack address]
...
lduh [stack address], %r[d']
...
stb %r[d'], [mem2]
```

In order to find these instruction sequences, phase 3 applies a simple instruction-scanning heuristic (see Figure 4.8) to the body of the candidate function that is successful in identifying both of these sequences. This heuristic has never failed with the vendor compilers that have been tested.

4.2.4 Phase 4: Verifying Instrumentation Sites

As shown in Table 4.1, phase 4 is responsible for finalizing the set of instrumentation sites for a given candidate function. It does this on a per-load-candidate basis. When a load candidate location is reached at runtime, a phase 4 trampoline is unconditionally branched to. Since the original load candidate instruction was in fact a register-indirect load, the source register can be examined to determine the effective load address. The phase 4 trampoline passes this value to the phase 4 function.

The first task performed by phase 4 is to look up this effective load address in the GMIT. If the effective address is found, a marker has been encountered, and the address of the load instruction corresponds directly to an instrumentation site. Whenever the address is found in the GMIT *and* there is at least one metric bound to the site, phase 4 replaces the load candidate instruction with an unconditional branch to an "instrumentation trampoline" that invokes all instrumentation bound to that site. If the effective address is not found in the GMIT, then the load candidate selected by phase 3 is a false positive. In this case, the original instruction(s) that led to the selection of the failed load candidate are replaced in the original program (which also removes the branch that transferred control to the phase 4 trampoline).

In both cases, execution resumes at the address of the load candidate, which is where control was initially transferred to the phase 4 trampoline. Thus, for real instrumentation sites, the next instruction executed will be the branch to the trampoline that executes the instrumentation for that site. It is worth nothing that phase 4 is applied exactly once to each load candidate site, so its cost is also amortized across the execution of the program.

One important feature of our implementation is that *the cost of executing instrumentation sites with no metrics bound to them is negligible.* This is because phase 4 never writes a branch to an instrumentation trampoline unless there are metrics bound to the site in question. This reduces the overhead incurred by the trampoline that invokes instrumentation for that site (see Section 5.1 for the results of the overhead experiments). If no metrics have been bound to a site when phase 4 is invoked, the load candidate instructions are overwritten with NOPs. Phase 4 then saves the relevant data pertaining to the site's location, and when a metric is dynamically bound to that site for the first time, these data are used to write the branch to that site's instrumentation trampoline. Thus, the only overhead incurred by the presence of instrumentation sites without bindings is the overhead of the NOP instructions, which is negligible. This feature of our implementation strategy is useful in practice because it allows the liberal placement of instrumentation sites within programs.

Figure 4.9 shows the code from Figure 4.7 after phase 4 been applied to both load candidates. In this particular example, the first load candidate turned out to be an instrumentation site, and so has been overwritten with a branch to the instrumentation trampoline. The latter `lduh` instruction, however, turned out to be a false positive and is restored to its original position.

4.2.4.1 Implementation Details

The method that our implementation uses to determine the effective load addresses is deserving of elaboration. When phase 4 trampolines are built by phase 3, the register-indirect load candidates instruction are known. On the Sparc V9 architecture, there are two kinds of register-indirect loads,

```
func:
    inst1
    inst2
    ...
    branch to instrumentation trampoline
    ...
    lduh [%o3], %o0
    ...
```

Figure 4.9: Candidate function after phase 4

each with a different way of computing the effective load address: (i) as the sum of a register value and an immediate offset, or (ii) as the sum of two register values.

To compute the effective address in the phase 4 trampoline, an `add` instruction (which mimics the corresponding register-indirect load candidate's address calculation) must first be written there by phase 3. When designing the structure of the phase 4 trampoline, we took great care not to prematurely clobber the register(s) holding components of the effective address. A standard trampoline will not work, because it issues a `save` before any other instructions. For phase 4 trampolines, the `add` must execute before the `save`. This is because the load candidate may use a local register for one of its source operands, and the contents of local registers are made inaccessible when a new register window is obtained by the `save`. Thus, the phase 4 trampoline prologue allocates one word on the stack to store the contents of the register that will be clobbered by the `add`, copies the value of the register to the stack, issues the `add`, and *then* issues the `save`. Likewise, the epilogue first issues a `restore` to restore the old register window, copies the saved value from the stack to the previously-clobbered register, and readjusts the stack pointer to the value it had in the candidate function.

4.2.5 Executing Monitoring Code and Invoking Metric Computation

Whenever an instrumentation site is reached during execution, all metrics currently bound to that site must be computed. This computation of metric values is invoked from instrumentation trampolines. These trampolines are activated whenever execution reaches an instrumentation site to which metrics are bound.

The computation of metrics bound to a particular site is straightforward. Each metric is associated with a metric function. Thus, each site maintains a list of metric functions to invoke, as well as a list of addresses where the return values of each invoked metric are to be stored. The dynamic binding mechanisms described in Section 3.6 directly manipulate this list of metric functions at runtime in order to add or remove[9] associations between metrics and the instrumentation sites to which they are bound. After saving the necessary execution state[10], a function is invoked by the instrumentation trampoline that iterates over this list of metric functions and invokes each one in turn. After all bound metric functions have been executed, the instrumentation trampoline restores execution state and transfers control back to the instrumented function in the application program.

Since branching to and from the instrumentation trampoline occurs at all instrumentation sites that have metrics bound to them, runtime overhead is incurred. There are two distinct kinds of overhead. The first type of overhead, called trampoline overhead, is the combined cost of three things: (i) transferring control to and from the trampoline, (ii) the execution of the trampoline itself, and (iii) the execution of the function which iterates over the list of metric functions and invokes them. Neither the cost of metric function invocation itself nor the cost of metric computation are accounted for in trampoline overhead. Rather, these costs are included in the second type of overhead, which is simply called the instrumentation overhead. Instrumentation overhead cannot be avoided when performance instrumentation is desired, and so it is not an express goal of the work presented here to minimize it. However, we measured the trampoline overhead incurred by our current implementation and the experimental results can be found in Section 5.1.

4.2.5.1 Implementation Details

The last section described how, at runtime, the dynamic binding mechanisms manipulate a list of bound metric functions that is associated with each site. This is an oversimplification of the roles played by a few key classes in our implementation of phase 4, the execution monitoring step, and the runtime library function (RTL) for the `bindMetric` language primitive. There are three classes that are responsible for collectively maintaining the dynamic state of the instrumentation sites: *InstInfo*, *InstSiteInfo*, and *InstFunctionInfo*, respectively. Phase 4, the execution monitoring

[9]Dynamic removal has not yet been implemented.

[10]This is vital because calls to arbitrary functions will be made, and since the vendor compiler did not witness these calls, it could not have constructed a valid register schedule for caller-saved registers.

step, and RTL function for `bindMetric` all act as clients of these classes. A discussion of these classes, their responsibilties, and their collaborations with each other follows. A UML diagram summarizing the three classes is found in Figure 4.10.

InstInfo is a singleton class (see [15]) that provides a convenient interface to data about the metrics that are bound to particular instrumentation points and intervals at runtime. It is responsible for mapping the unique identifier (i.e., the name) associated with an instrumentation point (or interval) to a pair of *InstSiteInfo* instances. In the case of intervals, each element in the pair represents the *InstSiteInfo* for one of the interval endpoints. For point sites, only the first element of the pair contains data. The *InstInfo* class is used by its clients to look up site information, provided to them in the form of *InstSiteInfo* instances.

An *InstSiteInfo* instance contains information about the dynamic state of a single instrumentation site. More specifically, it holds information about the branch to the site's instrumentation trampoline, and a list of (pointers to) *InstFunctionInfo* instances that contain information about the metric functions themselves. The *InstSiteInfo* class supports three key operations: (i) registration of new metric functions at the site via the `push_back` method, (ii) invocation of all registered metric functions via the `invokeFunctions` method, and (iii) installation of a branch to an instrumentation trampoline via the `installBranch` method. The first operation is used by the RTL function that implements the `bindMetric` directive, and forms a new association between a metric function and the *InstSiteInfo* instance's instrumentation site. The second operation, `invokeFunctions`, is called from the instrumentation trampoline for an instrumentation site, and iterates over the `instFuncInfos` list, calling the `invoke` member function on each element. Finally, the third operation, `installBranch`, is invoked by both phase 4 and by the `push_back` method. At the time of invocation from phase 4, the `installBranch` method decides not to install the branch if no metrics have been bound to the instrumentation site (the reasons for this are described in Section 4.2.4).

The *InstFunctionInfo* class is used to represent information about a single metric function *at a particular site*. This information consists of: the address where the function's return value should be stored, a pointer to the metric function, whether or not this function has been invoked at the site, and a link to another *InstFunctionInfo* instance[11]. Calling the `invoke` member function on an

[11]This link is used by *InstFunctionInfo* instances that represent end sites of instrumentation intervals to find the matching start site's *InstFunctionInfo* instance.

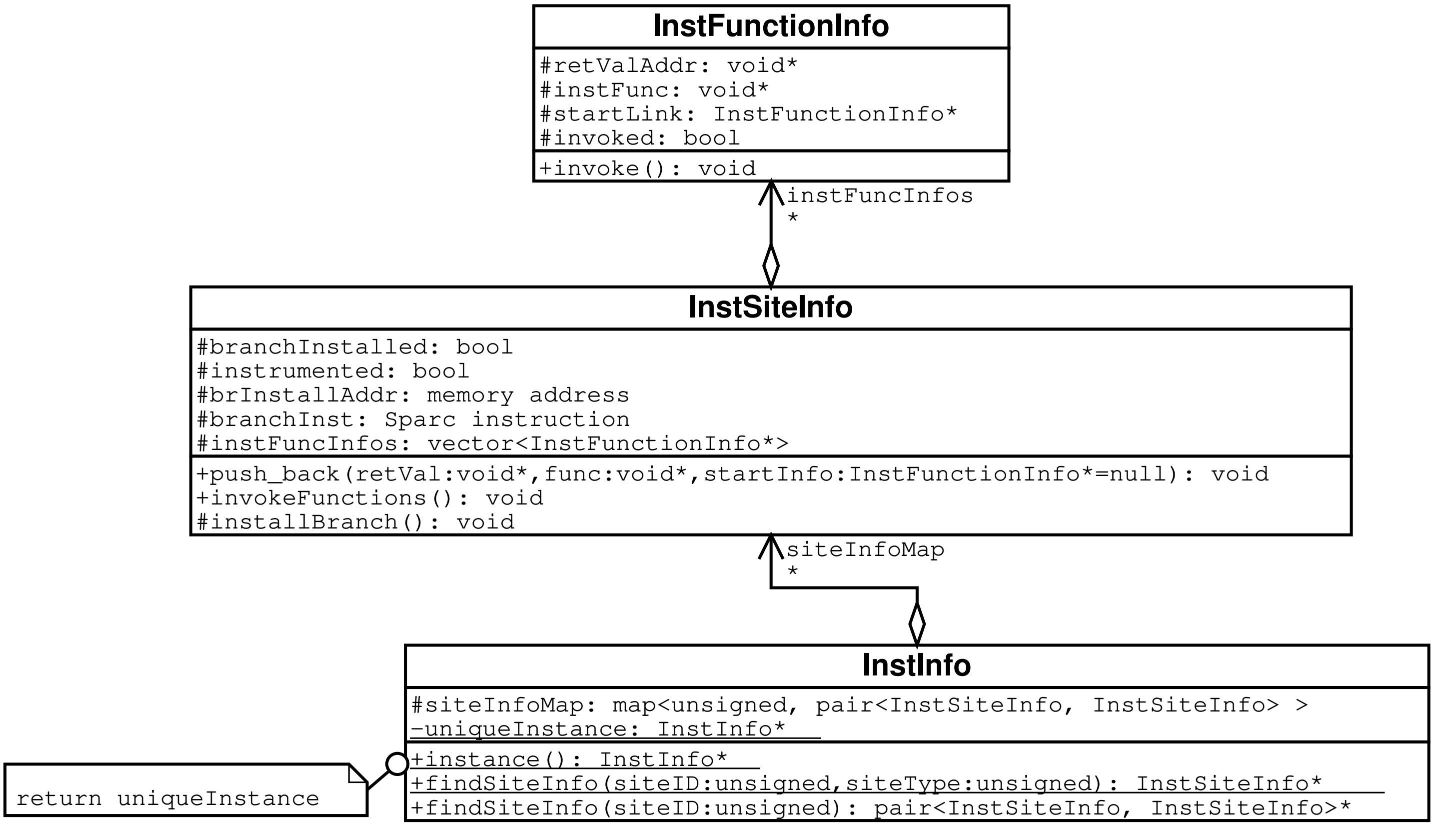

Figure 4.10: UML class diagram depicting the site management classes

InstFunctionInfo instance calls the function pointed to by `instFunc`, and saves its return value at the address given by `retValAddr`.

One important feature of the *InstFunctionInfo* class is the flag that denotes whether the metric function has been invoked. This flag is used to ensure that the metric functions invoked at end-interval sites are matched by prior calls to the corresponding metric functions at the start site for the same interval (i.e., if the flag for a start-site metric function is not set, the corresponding end-site function is not invoked). For a particular interval, the flag is set in the start site's *InstFunctionInfo* instance when `invoke` is called on it, and cleared when `invoke` is called on the *InstFunctionInfo* instance for the end site. These flags are needed because the `bindMetric` primitive can bind new metrics to same interval that it resides in — without these flags in place, instrumentation interval semantics break because metric functions at end sites of intervals can be executed before the corresponding functions at the start sites.

Chapter 5

Experimental Results

In this chapter, experimental results are presented for the overhead of the implementation of the runtime phases, followed by a description our experience with describing performance properties in two application codes. A brief summary of findings is provided in Section 5.3.

In order to evaluate the design of the language extensions for performance properties and our implementation, this section addresses the following questions experimentally:

1. Does the multi-phase runtime approach taken by this thesis (in order to attain vendor independence) incur significant runtime overhead, i.e., is there a significant performance penalty to using the presented approach over and above the cost of the instrumentation itself?

2. What are the benefits of using the proposed language extensions to describe the performance properties of applications for which performance is a primary concern? This question is only discussed subjectively and based on our experience with the language extensions in two applications.

In addressing question 2 above, it should be emphasized that it is not the goal of the experiments to evaluate the efficacy of a particular description of an application's performance properties. However, the attempt is made to assume the role of the application developer and describe performance properties that are cogent and as application-specific as possible.

Benchmark	orig	mark	p2	p2-3	p2-4	all
em3d	11.91s	1.26%	1.93%	2.44%	2.1%	3.69%
tsp	16.25s	0.55%	0.8%	1.85%	1.48%	1.72%
health	17.03s	0.59%	1.76%	1.88%	2.11%	3.52%
voronoi	26.07s	0.57%	1.11%	0.72%	1.83%	2.22%
power	255.44s	0.25%	0.25%	0.34%	0.25%	0.7%

Table 5.1: Overhead incurred by runtime phases

5.1 Overhead Experiments

In order to evaluate the runtime overhead incurred by the multi-phase approach, it is important to recognize the sources of overhead (the runtime phases themselves) and what kind of overhead to measure. The kind of overhead that needs to be measured is important because it should not include the cost of performing metric computation; i.e., the cost of any measurement functions that are invoked by the instrumented program should not be considered as overhead. In order to realize this in the experiment setup, the monitoring step was modified so that while branches to the instrumentation trampolines are still written by phase 4, the instrumentation trampolines do not actually invoke the measurement functions themselves.

The rest of the experimental setup is as follows. Five Olden [10] benchmarks were profiled to determine the functions in each where the most execution time was spent. An instrumentation interval was placed over the body of this most-executed function, or in a loop nest that invoked the function. The intention of this profile-guided placement of instrumentation intervals is to place markers and register instrumentation that will be executed frequently, so as to better evaluate the incurred overhead. After placing an interval in each benchmark code, a single metric is statically bound to the interval (it does not matter which metric, since the cost of doing the sampling is not taken into account). Each benchmark is timed for each runtime phase in succession[1]. That is, the execution times were taken for the benchmark with: no modifications ("orig" in the Table 5.1), with only the instrumentation markers in place ("mark"), with only phase 2 enabled ("p2"), with only phases 2-3 enabled ("p2-3"), with only phases 2-4 enabled ("p2-4"), and with phases 2-4 enabled together with the modified monitoring step ("all"). In Table 5.1, each execution time (other than

[1] In the current implementation, any runtime phase can be disabled by allowing each prior phase to perform all of its responsibilities except for writing the branch instruction that will eventually invoke the trampoline to latter phases.

for the original) is given as a percentage increase over the execution time of the original benchmark.

To consider a worst-case scenario, instrumentation was placed in a small, inlinable routine in the voronoi benchmark. The routine is executed millions of times from within the context of a larger routine that constitutes nearly 92% of the overall execution time. Because of this, the presence of the instructions for the markers incurs a larger overhead cost than in a typical instrumentation scenario. Still, it was discovered that the overall overhead (i.e., the value of the "all" column for this worse-case experiment) was around 15%. This cost reflects the presence of branches to and from instrumentation trampolines from a function on the critical path. This cost is not altogether unexpected, however, given the nature of the instrumented function – in general, the ability to instrument very small, frequently-executed routines without incurring high runtime costs is not a reasonable expectation.

As can be seen in Table 5.1, the runtime overhead for normally-instrumented benchmarks is considerably small. What these numbers mean, in practice, is that application codes can be richly instrumentable (i.e., contain many instrumentation sites), with only small overheads incurred before the actual instrumentation functions are applied. For the instrumentation sites that have no metrics bound to them, the monitoring step is never invoked, and so the "p2-4" column in Figure 5.1 reflects the overhead that would be experienced.

5.2 Applications

Two applications are used for our experiments: POV-Ray and a distributed proxy server application. Aspects of each each application have been identified that are considered to be relevant to its performance, based on our knowledge of the application. These performance properties are then encoded in the applications themselves using the proposed language extensions.

5.2.1 POV-Ray

POV-Ray is a widely used, publicly available raytracing application [30]. As such, it is a compute- and memory-bound application. This means that aspects of memory performance such as TLB misses, L1 and L2 cache access patterns, and load and store misses are of primary concern. With this in mind, the goal is to assume the role of the application developer in an attempt to learn something about the performance characteristics exhibited by POV-Ray. Some of the features supplied by the proposed language extensions will be used in order to accomplish this goal.

One common problem facing performance tuners is the act of correlating different types of performance data when a degradation in performance is experienced. The typical approach requires recording large volumes of data, and these are typically examined offline whenever a correlation between a high-level performance degradation and potential lower-level causes is desired. However, forming such correlations is not even possible in many cases.

In contrast to the typical approach, the performance properties and language mechanisms allow the examination of detailed performance data only when performance actually degrades, and this examination can be accomplished online. This goal is achieved by using two mechanisms exposed by the language extensions, namely *performance assertions* [38, 29] and *self-monitoring of performance data*.

The first step in making these kinds of performance correlations in POV-Ray is to identify where potential correlation data are to be gathered in the application, and what these datasets should contain. We profiled POV-Ray and determined that the routine `trace_pixel` consumes more than 76% of its computation time (for a benchmark scene), so it is a natural choice for instrumentation[2]. *The key question we ask with this experiment is*, "When `trace_pixel` is slower than average, what other metrics are high?"

To answer this question, we chose elapsed time as the high-level metric, and some aspects of cache behavior as potential low-level causes. At the top of the code in Appendix A, three metrics are declared: `elapsedTimeSeries`, `L1ICacheMisses`, `L2TCacheMisses`. These metrics represent a moving average of elapsed time, an L1 instruction cache miss count, and an L2 total cache miss count, respectively[3]. All are statically bound (see the `declareInterval` directive at the start of the sample scope) to interval `i1`, which is defined over the body of the `trace_pixel` routine. Thus, data pertaining to memory characteristics are gathered on a per-invocation basis (i.e., each time `i1` is executed, samples from previous `trace_pixel/i1` invocations are discarded), and the elapsed

[2]Note that if our language primitives were a bit more mature (e.g., if they provided MDL-like "globbing" mechanisms to specify binding to groups of intervals), using a profiling tool should not have been needed. Rather, we should have extended the application logic and let *it* determine the best routine to analyze.

[3]Due to a lack of available low-level metrics at the time of implementation, we were unable to compute other important metrics such as L1 data cache or TLB misses.

time for `i1` is computed and averaged over a moving window.

Now that it is known what kinds of data to collect, it must be decided what we mean by the term "performance degradation" vìs-a-vìs our key question . For the code in Appendix A, performance degradation is considered to have occurred anytime that the measured elapsed time for a particular instance of interval `i1` exceeds the moving average by some fixed "badness" factor. In order to encode this property, the language primitives were used to add a performance assertion to the code. In the example code, the performance assertion is found after the sample interval. The most recent elapsed time sample is compared against the moving average (see the lines immediately following the calls to functions `pp_series_last` and `pp_avg`). If the most recent sample exceeds the average by a factor of `BADPERF_THRESHOLD`, the memory behavior samples for the *current* invocation of the routine are reported. Thus, the application itself can provide an online correlation between the experienced performance degradation and potential low-level causes. Since the detailed data are only reported or recorded when a performance degradation is actually witnessed, the typical scenario wherein exorbitant amounts of irrelevant data are collected ceases to be a problem.

The fact that the above performance properties could be expressed in the POV-Ray application using only a modicum of new statements (see Appendix A) in the original application code testifies that the proposed language primitives are usable in practice.

One of our experiments showed that POV-Ray exhibits interesting L2 cache behavior for a benchmark scene (see Figure 5.1). We slightly modified the code found in Appendix A so that it maintained a moving average of L2 total cache misses in addition to the moving average of elapsed time. It also maintains an additional metric variable named `al2` with accumulator type `sumcount`, which is used to compute the average of the "miss-over-average-miss" ratio $\frac{L2TotalMiss}{L2TotalMissAvg}$, where $L2TotalMiss$ is the most recent sample of L2 total cache misses and $L2TotalMissAvg$ is the moving average value.

Whenever the performance assertion is violated, `al2` is sampled, meaning that the above ratio is computed and accumulated into `al2`'s implicit accumulator, and `al2`'s implicit counter is incremented. The average of the "miss-over-average-miss" ratio is finally queried when the entire scene has finished rendering. These averaged ratios are shown (as percentage increases) on the y-axis of Figure 5.1. The x-axis represents the value of `BADPERF_THRESHOLD` when the performance assertions

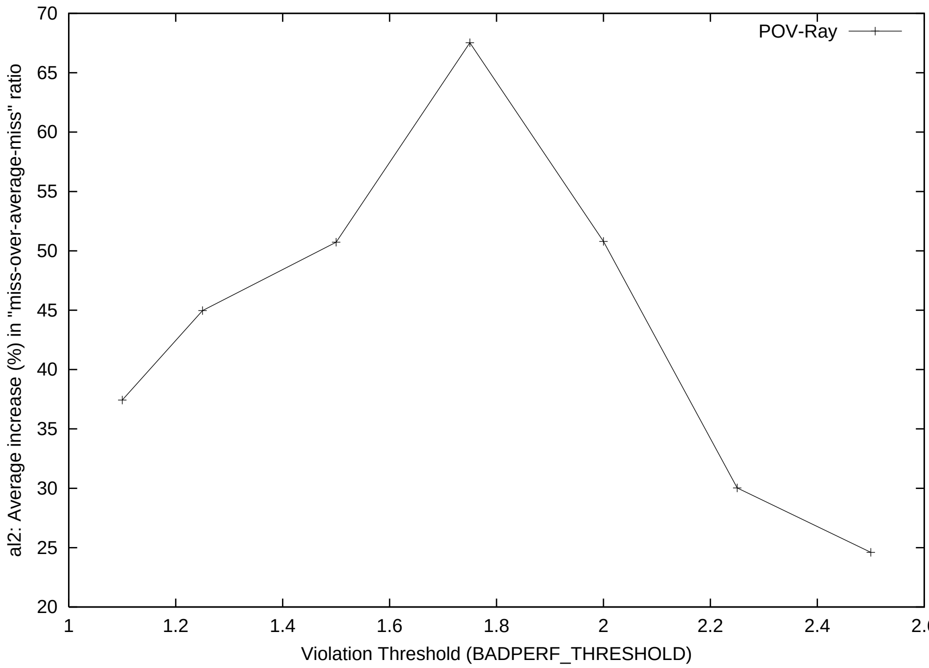

Figure 5.1: L2 Total Cache Miss Behavior in POV-Ray

were triggered. The graph indicates a potential correlation between the severity of the performance degradation that occurred, and the increase in L2 total cache misses that were witnessed at the same time — at least up until a certain point. After a `BADPERF_THRESHOLD` value of about 1.8, the queried value of `al2` decreases rapidly. Although we are not certain of the reason(s) for this drop-off, we do know that when the violation threshold is that high it is violated for roughly less than 1.74% of the pixels in the entire image. Computational workloads in raytracing applications are rarely homogeneous across pixels, and so this drop in the "miss-over-average-miss" ratio could simply indicate real CPU cycles being spent.

The main point of this experiment is to illustrate the kinds of complex performance aspects that can be encoded into applications using our proposed language extensions. All of the data for Figure 5.1 were generated online, and furthermore the `al2` sampling *only takes place when performance degradation is witnessed*. The `al2` queries are recorded after the image is complete. Thus, no extraneous data are recorded.

Of course, other performance properties or performance monitoring actions could be chosen. For example, the application programmer might decide to record queried metric values in a table, indexed by pixel or image quadrant, to determine if particular parts of the image were causing aberrant memory behavior. Or, the programmer could choose to have certain rendering modes adaptively disabled if too many performance degradations were witnessed within a certain period of time (this may actually occur in an interactive raytracing program). We believe that some of these possibilities are among the most impressive aspects of the system proposed in this thesis.

In summary, our experiments with POV-Ray showed that the language primitives can be used to engage in performance-oriented programming. Also, when coupled with the regular control structures of the source language, these primitives allow the expression of considerably complex performance properties. Finally, the language mechanisms address some common problems (e.g., the correlation problem described above) with the analysis of data gathered by typical performance instrumentation methods.

5.2.2 Distributed Proxy Server

For this experiment, a lightweight proxy server is used which performs on-the-fly image distillation for its clients, and decides the extent to which the images from the web should be degraded based on the available bandwidth. The code was developed in our group, and is representative of many long-running server type applications that could benefit greatly from being able to monitor their own performance and especially to identify anomalous performance conditions at runtime.

In our experimental scenario, the proxy server application is executed under normal operating circumstances (i.e., moderate client load), and the elapsed time for performing a generic distillation operation is measured and averaged across the interval `i1`. If any performance degradations occur (i.e., the performance assertion which checks the value of the elapsed time average is triggered), the language primitives for dynamic binding are used to add a new metric (total L2 cache misses) to sampling intervals `i2` and `i3`, both of which are subintervals of `i1`. The idea is that when image distillation is "taking too long", new metrics are bound and computed at a finer grain. The relevant code for this application, which shows the placement of performance-oriented constructs, can be found in Appendix B.

To describe the performance assertion and the logic that dynamically binds new interval metrics when the assertion is violated, only a few lines of code are added to the server application. At the top of the code, an interval metric called `elapsedTimeSeries` is declared, which maintains a bounded series of elapsed times for the interval of code which performs the actual image manipulation. The average value of such a series is the kind of performance property that a programmer may wish to make an assertion about. Next, braces are added in order define the scopes containing the intervals to be measured, and `declareInterval` statements are placed inside those scopes to specify them as instrumentable intervals. The `elapsedTimeSeries` metric is the only statically-bound metric, and it is bound to interval `i1`. Finally, a performance assertion is used to specify that the average of the series should be below a specified threshold.

When the performance assertion is violated, the `bindMetric` directives (in the `if` statement body that corresponds to the performance assertion) are used to bind the `L2CacheAccesses` metric to two other intervals (`i2` and `i3`) that previously had no metrics associated with them; upon subsequent executions of those intervals, the number of L2 cache misses that occur will be sampled and aggregated. The performance assertion and its associated code appear at the bottom of the function, where the moving average is queried using the `pp_avg` function. Because, after transformation, the metric variables are simply language variables, and the statistical functions are function calls, describing complex performance assertions becomes a straightforward and natural process.

In conclusion, it is worth nothing that that there are a host of other performance properties which may merit monitoring in a server code such as this. In particular, the programmer may wish to monitor bandwidth fluctuations, server throughput, server CPU load, transactions completed per second, number of active client connections over a period of time, or a custom metric such as "standard deviation over a fixed window size of active client connections divided by average server throughput, using data aggregated from three selected execution intervals". All of these metrics can be described in a straightforward manner. With little extra implementation effort, the scalability properties of the application could be assessed by correlating current (dynamic) performance data with saved data about how the application had performed under smaller loads or on fewer processors.

5.3 Summary

In Section 5.1, it was shown that the runtime overhead incurred by our approach in the Olden benchmarks was acceptable. This means that the language extensions proposed in this thesis can be used in practice. In Section 5.2, furthermore, the suitability of these language extensions for describing complex performance properties in serious applications was clearly demonstrated.

For POV-Ray, a solution to a common problem was presented — the correlation between high-level performance degradation and its potential causes. This problem is difficult to solve online without the ability to describe performance assertions, and impossible to solve online without language mechanisms that permit self-monitoring of performance data. The proposed language extensions permit both of these to occur in a straightforward manner.

The distributed proxy server was used to demonstrate the power of having the associations between metrics and instrumentation sites be first-class semantic properties of the language. In particular, dynamic control over instrumentation becomes possible and is useful in practice.

In both applications, all metrics and performance assertions are encoded as intrinsic properties of the program, and thus are "carried" with it as it is built and used on different platforms. Provided that the runtime system exists on a particular target platform, this approach effectively provides portable performance monitoring without additional effort on the part of the application programmer.

Finally, the code in both appendices demonstrates the ease with which the language extensions are able to realize relatively complex performance monitoring concepts such as performance assertions and self-monitoring of performance data.

Chapter 6

Conclusion

This thesis presented a small set of language extensions for describing, in a flexible and succinct manner, performance properties of application codes at the language level. It has also presented a novel implementation scheme that focuses on reducing perturbation due to inhibited compiler optimizations, and that is usable in practice because it is vendor-compiler independent.

The primary strengths of the work presented in this thesis are as follows: first, we have described how the performance features of applications can be thought of as intrinsic properties of the code. Second, we have delineated the flexibility of our language extensions by describing their potential high-level uses in performance-critical applications (e.g., self-monitoring of performance and performance assertions). Third, we have described how our implementation strategy permits a plethora of performance-related actions (e.g., data aggregation schemas, user-defined metrics and sampling intervals, and dynamic control of instrumentation) to be specified at statement- and scope-level granularity. Finally, we have presented concrete examples wherein performance properties are encoded into real applications to good effect.

The work adduced in this thesis has the following weaknesses: first, the lack of a rich library of metric functions on the Sparc platform[1] prevented us from fully demonstrating (what we consider to be) the power of our language-based approach to performance instrumentation. Second, the language extensions themselves are immature; e.g., the primitives currently provide no *clean* way to build composite metrics, nor can instrumentation intervals and sites be classified or grouped in any way. Finally, although we have presented a system that is intended to be practical, we

[1]Although PAPI provided a number of useful metrics, key metrics such as TLB misses or L1 data cache misses seemed not to be supported. Also, we lacked a mechanism to disambiguate total cycle counts into CPU and memory cycles.

lack an objective evaluation criteria for determining whether or not our approach could *actually* be integrated into modern software development practices — although we certainly believe that it could.

Besides modifying the existing system in order to address the weaknesses described above, interesting future work falls into two broad categories: (i) extending the system to realize more complicated performance-oriented goals, such as semi-automatic performance prediction or automated bottleneck detection, and (ii) attempting to find more useful language primitives that can facilitate the widespread adoption of our approach and "bridge the gap" betwixt high-level application concerns and low-level architectural and system constraints.

In conclusion, this thesis has presented a new way of looking at the performance instrumentation of modern software applications, and provided a solid basis for so-called "performance-oriented programming", which has the potential to unify aspects of traditional software development phases (e.g., application development and its currently-disjoint successor, performance debugging) and drastically change the way that the applications of tomorrow are developed.

References

[1] A. M. Alkindi, D. J. Kerbyson, and G. R. Nudd. Dynamic instrumentation and performance prediction of application execution. *Lecture Notes in Computer Science*, 2110:513–??, 2001.

[2] A. M. Alkindi, D. J. Kerbyson, E. Papaefstathiou, and G. R. Nudd. Optimisation of application execution on dynamic systems. *Future Generation Computer Systems*, 17(8):941–949, 2001.

[3] Sameer Shende Allen. Integrated tool capabilities for performance instrumentation and measurement.

[4] 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.

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

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[7] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[8] Bryan R. Buck and Jeffrey K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. pages 64–65, 2000.

[9] Harold W. Cain, Barton P. Miller, and Brian J. N. Wylie. A callgraph-based search strategy for automated performance diagnosis (distinguished paper). *Lecture Notes in Computer Science*, 1900:108–??, 2001.

[10] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. Technical Report TR-483-95, Princeton University, 1995.

[11] Ptools Consortium. PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/projects/papi/, 2003.

[12] Dawson R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[13] D.J. Kerbyson et al. PACE: A toolset to investigate and predict performance in parallel systems. *Presented in European Parallel Tools Meeting, ONERA, Paris*, October 1996.

[14] Dongarra et al. Experiences and lessons learned with a portable interface to hardware performance counters.

[15] Gamma et al. Design patterns: Elements of reusable object-oriented software, 1995.

[16] Romer et al. Instrumentation and optimization of win32/intel executables using etch. In *USENIX Windows NT Workshop*, pages 1–7, 1997.

[17] 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.

[18] Jeffrey K. Hollingsworth. Finding Bottlenecks in Large Scale Parallel Programs. Technical Report CS-TR-94-1243, September 1994.

[19] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *International Conference on Supercomputing*, pages 185–194, 1993.

[20] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, 1994.

[21] Jeffrey K. Hollingsworth, Barton P. Miller, M. J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *IEEE PACT*, pages 201–, 1997.

[22] Karen Karavanic and Barton Miller. Improving online performance diagnosis by the use of historical performance data. pages ??–??, 1999.

[23] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

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

[25] 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.

[26] 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.

[27] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *International Conference on Computational Science (2)*, pages 904–912, 2002.

[28] S. Muchnick. Advanced compiler design and implementation, 1997.

[29] Sharon E. Perl and William E. Weihl. Performance assertion checking. In *Symposium on Operating Systems Principles*, pages 134–145, 1993.

[30] Povray. http://www.povray.org.

[31] Todd A. Proebsting and Benjamin G. Zorn. Tangible program histories. Technical Report MSR-TR-2000-54, Microsoft Research, May 2000.

[32] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[33] Daniel A. Reed. Performance instrumentation techniques for parallel systems. In *Performance/SIGMETRICS Tutorials*, pages 463–490, 1993.

[34] John F. Reiser and Joseph P. Skudlarek. Program profiling problems, and a solution via machine language rewriting. *ACM SIGPLAN Notices*, 29(1):37–45, 1994.

[35] SGI. SGI SpeedShop performance analyzer. http://www.sgi.com/developers/devtools/tools/speedshop.html, 2003.

[36] Sameer Shende, Allen D. Malony, and Steven T. Hackstadt. Dynamic performance callstack sampling: Merging TAU and DAQV. In *PARA*, pages 515–520, 1998.

[37] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[38] Jeffrey S. Vetter and Patrick H. Worley. Asserting performance expectations. *Presented at SC 2002*, 2002.

Appendix A: Code Segment of the POV-Ray Application

```
/* The metric declarations for the elapsed time series and the two cache metrics. */
pp_interval<bounded_series, elapsed_time_start, elapsed_time_end, size=1000> elapsedTimeSeries;
pp_interval<scalar, l1_icache_misses_start, l1_icache_misses_end>            L1ICacheMisses;
pp_interval<scalar, l2_total_misses_start, l2_total_misses_end>              L2TCacheMisses;

/* "Bad" performance threshold, as a fractional increase over the averaged elapsed time.
    If the last sample in the elapsed time series exceeds the average by more than a
    factor of BADPERF_THRESHOLD, it is considered bad performance */
#define BADPERF_THRESHOLD 1.25

static void trace_pixel(int x, int  y, COLOUR Colour)
{
    { /*---- BEGIN SAMPLE SCOPE ----*/

        /* All three of our metrics are statically bound to this interval, i1 */
        declareInterval i1(elapsedTimeSeries, L1ICacheMisses, L2TCacheMisses);

        Increase_Counter(stats[Number_Of_Pixels]);
        Trace_Level = 1;
        POV_PRE_PIXEL (x, y, Colour);
        COOPERATE_0;

        /* Do histogram stuff */
        if (opts.histogram_on)
          accumulate_histogram(x, y, TRUE);
        if (Focal_Blur_Is_Used) {
            /* Use focal blur tracing. */
          focal_blur(&Camera_Ray, Colour, (DBL)x, (DBL)y);
        }
        else {
            /* Create and trace ray. */
            if (create_ray(&Camera_Ray, (DBL)x, (DBL)y, 0)) {
                Increase_Counter(stats[Number_Of_Samples]);

                if (opts.Options & USE_VISTA_BUFFER)
                    Trace_Primary_Ray(&Camera_Ray, Colour, 1.0, x);
                else
                    Trace(&Camera_Ray, Colour, 1.0);
            }
```

```
            else
                Make_ColourA(Colour, 0.0, 0.0, 0.0, 0.0, 1.0);
        }
        if (opts.histogram_on)
            accumulate_histogram(x, y, FALSE);
        Clip_Colour(Colour, Colour);
        gamma_correct(Colour);
     } /*---- END SAMPLE SCOPE ----*/

    /* The performance assertion */
    {
        double lastSample = pp_series_last(elapsedTimeSeries);
        double seriesAvg = pp_avg(elapsedTimeSeries);

        if(lastSample > seriesAvg * BADPERF_THRESHOLD) {

          /* Bad performance has been noted, so report the other metrics that
             were sampled for this particular invocation of trace_pixel */

          reportCacheMetrics(L1ICacheMisses, L2TCacheMisses);
        }
    }
}
```

Appendix B: Code Segment of a Distributed Proxy Server

```
pp_interval<bounded_series, elapsed_time_start, elapsed_time_end, size=20> elapsedTimeSeries;
pp_interval<bounded_series, l2cache_accesses_start, l2cache_accesses_end, size=5> L2CacheAccesses;

/* PA threshold is 1.5 seconds */
#define WARNING_THRESHOLD 1.5

#include <pp.h>
#include "fileManipulator.h"
#include "filesEditor.h"
#define JPEG 0
#define GIF 1
#define PS 2
#define PDF 3
#define FAILED -1
#define OKAY 1

extern char* contentLength[2];
extern char* contentTypes[2];
extern char* types[4];

int editFile(char** buffer, int bufSize, tableEntry* entry)
{
    int error;
    int size = bufSize;
    int type;
    int lengthKind = 0;
    int typeKind = 0;
    int flag = OKAY;

    char* contentType = NULL;
    char* contentSize = NULL;
    char* origFileName = NULL;
    char* split;

    origFileName = (char *)malloc(L_tmpnam);

    typeKind = getContentType(*buffer, &contentType);
    if(typeKind < 0) {
      flag = FAILED;
```

```
  printf("Error(editFile): Cannot find content type.\n");
}

if (flag == OKAY) {
  type = contentToInt(contentType);

  if (type < 4) {
    lengthKind = getContentSize(*buffer, &size, &contentSize);
    if (lengthKind >= 0) {
      /* write the buffer into a file so that it can be edited using
         the system() call */
      error = writeToFile(*buffer, size, type, &origFileName);
      if(error < 0) {
        flag = FAILED;
        size = bufSize;
      }
    }

    { /*---- BEGIN SAMPLE SCOPE ----*/
      declareInterval i1(elapsedTimeSeries);

      if (flag == OKAY) {
        switch(type) {
          case JPEG:
          {
            { /* ---- BEGIN SAMPLE SCOPE ---- */
              declareInterval i2;
              error = compressJPEG(calcLevel(entry->bandwidth, size), origFileName);
            } /* ---- END SAMPLE SCOPE ---- */

            if (error < 0) {
              flag = FAILED;
              size = bufSize;
            }
            break;
          }
          case GIF:
            break;
          case PS:
          case PDF:
          {
            { /* ---- BEGIN SAMPLE SCOPE --- */
```

```
          declareInterval i3;
          psdfToText(origFileName);
          error = changeHeader(buffer, contentTypes[typeKind], "text/plain", bufSize);
        } /* ---- END SAMPLE SCOPE ---- */

        if (error < 0) {
          flag = FAILED;
          size = bufSize;
        }
        break;
      }

      default:
        size = bufSize;
        flag = FAILED;
        break;
    }
  }
} /*---- END SAMPLE SCOPE ----*/

if(flag == OKAY) {
  /* read the file into the buffer after it has been modified.  returns the number
     of bytes after the header in the buffer */
  error = readFileToBuffer(buffer, origFileName, &size);
  if (error < 0) {
    flag = FAILED;
    size = bufSize;
  }
}

if(flag == OKAY) {
  /* change the header line Content-Length: to reflect the change of file size */
  sprintf(contentSize, "%d", error);
  error = changeHeader(buffer, contentLength[lengthKind], contentSize, size);
  if (error < 0) {
    flag = FAILED;
    size = bufSize;
  }
}
}
}
```

```
    /* remove the file since we don't need it */
    remove(origFileName);
    free(contentType);
    free(contentSize);
    free(origFileName);

    if(pp_avg(elapsedTimeSeries) > WARNING_THRESHOLD) {
      /* Performance assertion has been violated, so
         bind the L2CacheAccesses metric to intervals i2 & i3 */

      bindMetric L2CacheAccesses to i2;
      bindMetric L2CacheAccesses to i3;
    }

    return size;
}
```