MDL: A Language and Compiler for Dynamic Program Instrumentation[1]

Jeffrey K. Hollingsworth[*] Barton P. Miller[+] Marcelo J. R. Gonçalves[+]
Oscar Naim[+] Zhichen Xu[+] Ling Zheng[+]
hollings@cs.umd.edu {bart,mjrg,naim,zhichen,lzheng}@cs.wisc.edu

[*]Computer Science Department
University of Maryland
College Park, MD 20742

[+]Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Abstract

We use a form of dynamic code generation, called dynamic instrumentation, to collect data about the execution of an application program. Dynamic instrumentation allows us to instrument running programs to collect performance and other types of information. The instrumentation code is generated incrementally and can be inserted and removed at any time. Our instrumentation currently runs on the SPARC, PA-RISC, Power2, Alpha, and x86 architectures. Specification of what data to collect are written in a specialized language called the Metric Description Language, that is part of the Paradyn Parallel Performance Tools. This language allows platform-independent descriptions of how to collect performance data. It also provides a concise way to specify how to constrain performance data to particular resources such as modules, procedures, nodes, files, or message channels (or combinations of these resources). We also describe the details of how we weave instrumentation into a running program.

1 Introduction

Dynamic (run-time) code generation is a powerful idiom that allows a system to adapt to changing functional demand and workloads. It has been used for extensible operating system kernels [15], to construct efficient network protocols [14], and for compile-on-demand for interpreted languages [3]. We use a form of dynamic code generation, called *dynamic instrumentation*, in the Paradyn Parallel Performance Tools [7,13] to make run-time decisions about what performance data to collect and when. Dynamic instrumentation differs from other run-time code generation schemes in that it periodically modifies a running program to collect information about its execution.

Instrumentation is the activity of collecting information about an execution without modifying the intent of the underlying calculation. Our code generation and modification techniques have a variety of uses, but in this paper we concentrate on their use in program instrumentation. We describe the MDL language, how to instrument running programs and analyze binary programs, our code generation scheme, and some performance measurements. The techniques described in this paper are part of the Paradyn tools, that run on Solaris (SPARC and x86), AIX, HP-UX, DEC Unix, and Windows/NT (x86). To permit using dynamic instrumentation as a foundation for constructing other run-time tools, we have developed an API [9] for run-time code insertion.

We have defined a language, called the Metric Description Language (MDL) to cleanly specify the data to be collected and how to collect it. The MDL is a specialized language that has two key roles. First, it specifies code to be inserted into the application program to calculate the value of performance metrics. This code includes simple control and data operations, plus the ability to instantiate and control real and virtual timers. Second, it specifies how the instrumentation code is inserted into the application program. This specification includes the points in the application program that are used to place the instrumentation code.

The technical challenges for this work include support for a wide class of architectures. We address this issue by using a standard language and intermediate form, and by keeping the instrumentation specifications simple. A second challenge is to generate efficient instrumentation code. While we cannot avoid instrumentation overhead, we work to minimize it. A third challenge is dealing with optimized code. Unlike many tools, we operate on code generated

1. This work is supported in part by Wright Laboratory Avionics Directorate (WLAD), Air Force Material Command, USAF, under grant F33615-94-1-1525 (ARPA order no. B550), and Department of Energy Grant DE-FG02-93ER25176. Hollingsworth was also supported by NSF award ASC-9703212. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of WLAD or the U.S. Government.

with the full optimization of current compilers. In some cases, this forces us to de-optimize small parts of code at run-time so that we can insert instrumentation. Our current instrumentation is at procedure granularity (entry, exit, call), so this complexity is manageable; future versions that handle basic blocks will require additional development.

Our dynamic code generation is more general than specialization, but it is not as general as dynamically compiling a procedural programming language. Since we incrementally generate relatively small pieces of code, we cannot afford the cost of a full compiler; but simple code templates based on specialization are too restrictive.

Dynamic instrumentation pushes the spectrum of instrumentation technology, complementing techniques such as binary rewriting [2,11,16,19]. Dynamic instrumentation defers the decision about what to instrument until program execution time. Performance can be evaluated on-the-fly and changes made to the instrumentation based on the application program's execution characteristics. This allows long-running programs (such as large scientific codes) and already-running programs (such as database servers) to be instrumented. Both dynamic instrumentation and rewriting have the advantage of not requiring access to the source code. Binary rewriting is a static process that works best when advanced knowledge about what to collect is available and remains fixed for the program's execution.

1.1 Instrumentation for Performance Debugging

Performance debugging of parallel programs requires a detailed understanding of the program's execution and its interaction with the hardware on which it is executing. Given the wide range of hardware, operating systems, programming languages, libraries, and primitives for parallelism, it is difficult to build performance measurement tools that can provide all the required detailed information for all hardware platforms. A key problem facing tool builders is how to create tools that are flexible enough to be useful on a variety of platforms, yet provide sufficient detail to assist the programmer.

Performance debugging consists of two steps: data collection and data presentation. In this paper, we focus on making the collection of new types of data easier. We present a language called MDL for describing parallel program performance metrics. MDL permits compiler, library, and even application programmers to customize a performance measurement tool to gather the desired data. Rather than continually adding hard-wired data collection and analysis techniques to a tool, it is preferable to build a performance tool that is extensible and permits the easy addition of new types of data. We had several goals for our metric description language:

Portability. Portability of a performance tool requires that the tool run on a variety of platforms, *and* that the instrumentation specifications be portable between platforms. For example, if a library writer describes a performance metric for a library, that description should be usable on any system that can run the library. To achieve this portability, data collection must be described at a higher level of abstraction than machine instructions. However, while it is important to have portable metrics, some types of data may not be relevant on particular platforms. Consequently, a metric description language needs to permit platform specific metrics.

Decoupling metrics from program components. Although metrics for the entire computation can be useful, generally it is necessary to gather data at a finer granularity. For example, it is possible to report I/O waiting time for an entire program, by file name, or by procedure. Depending on the situation, any of these might be useful, or even a combination of them. Describing individual metrics for all the different ways to isolate the data is not practical. If there are n metrics, and m orthogonal ways to constrain those metrics, then there are $n2^m$ different combinations! Writing custom definitions for each metric and program component combination is impractical for more than a few metrics and program components. We decouple the description of a metric from how to constrain it to different program components. As a result, it is sufficient to describe m constraints and n metrics ($m+n$).

The rest of this paper describes the language and its implementation. Section 2 describes the MDL language. Section 3 describes how we obtain structural information from the binary executable. Section 4 describes how we insert and remove the code from a running program. Both Section 3 and Section 4 discuss issues relating to instrumenting optimized code. Section 5 describes our dynamic instrumentation and code generation.

2 Metric Description Language (MDL)

MDL is a special purpose language for writing instrumentation requests. Instrumentation requests are written in terms of *performance metrics*. A performance metric is a time-varying function that characterizes some part of a program's behavior (such as percent CPU usage or message bytes per second). Specifications written in MDL describe the basic instrumentation to calculate a performance metric and additional specifications for how to constrain the metric to different program components. An MDL metric description represents a potentially enormous number of variations of possible performance measurements to gather. However, only those metric combinations that have been requested will execute.

An MDL description can be thought of as a two-part program. The first part describes where to insert the program instrumentation code (the *where* specification); the second specifies the code that will be inserted into the application (the *what* specification). When, during execution, a request is made to instrument the application program, the where specification is interpreted to compute the places in the application to instrument. The what specification is then translated into machine code and combined with the where-specification and information about the control structure of the application program. The machine code is then inserted into the running program. Figure 1 shows the overall flow of information during MDL evaluation

Below, we introduce our model of program instrumentation and describe the MDL language. At the end of the section, we present a short example that shows how an MDL metric description is combined with a request to generate instrumentation.

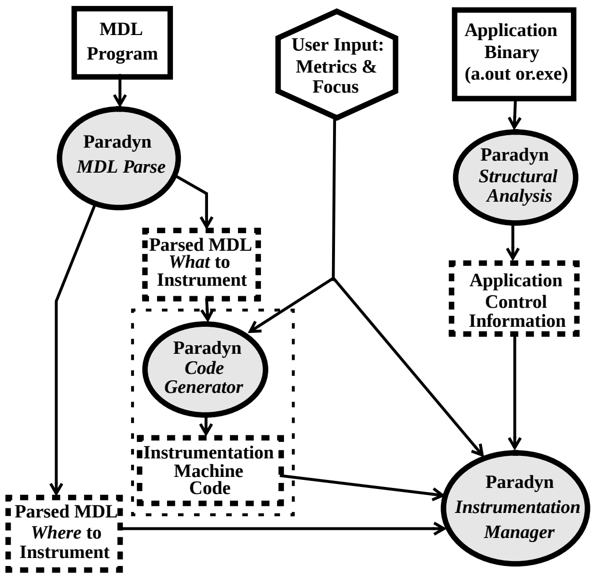
MDL Program
User Input: Metrics & Focus
Application Binary (a.out or.exe)
Paradyn MDL Parse
Paradyn Structural Analysis
Parsed MDL What to Instrument
Application Control Information
Paradyn Code Generator
Instrumentation Machine Code
Paradyn Instrumentation Manager
Parsed MDL Where to Instrument

Figure 1: MDL Flow of Control
Dashed-line boxes are generated by Paradyn

2.1 Introduction to MDL

Previously, we developed a simple, well-defined set of operations that can be used as building blocks to compute metrics for the desired program components [7]. To collect data, we insert software instrumentation into the program. By keeping the instrumentation operations simple, we can optimize their performance for each platform. Recording performance information about the application program is accomplished by *points*, *primitives*, and *predicates*. Points are well-defined locations in the application's code where instrumentation can be inserted. Currently, the available points are procedure entry, procedure exit, and individual call statements. In the future, points will be extended to include basic blocks and individual statements. Primitives are simple operations that change the value of a counter or a timer. Predicates are Boolean expressions that can be associated with primitives to determine if the associated primitive gets executed. By inserting predicates and primitives at the correct points in a program, a wide variety of metrics can be computed.

MDL differs from most languages in that part of the program specification executes when a request to insert instrumentation is received, and part of it executes inside the application process to measure its performance. To clarify the two parts of the language, consider this example:

```
foreach i in $procedures
    append preInsn i.entry
     (* proc_calls++; *)
```

The first line contains a loop that (implicitly) declares a variable to iterate through the list of procedures defined in the list variable `$procedures`.[2] The second line contains two keywords, `append` and `preInsn` (described below) that indicate where the instrumentation code should be inserted. The second line also defines the point where the instrumentation should be inserted. In this example, we are inserting the instrumentation at the beginning of the entry to the procedure defined by the variable `i`. The instrumentation code to insert is bracketed by the tokens **`(*`** and **`*)`**. The code inserted will increment a variable `proc_calls`. The result of this code snippet is that every time a procedure is called in the application program, the counter variable `proc_calls` is incremented by one.

When instrumenting a point, controlling whether the instrumentation happens before or after the code is desirable, as is controlling the order of snippet execution if multiple snippets are inserted at a single point. MDL includes two parameters for controlling the placement of instrumentation code. We can control whether instrumentation is executed before or after the instruction corresponding to the instrumentation point. This is especially useful for points that are procedure calls, since it permits instrumentation to be inserted either immediately before the call, or immediately after it returns. The two possible values for this modifier are `preInsn` and `postInsn`. We can also control the order of instrumentation code snippets if more that one snippet is inserted at a given point. When a new snippet is to be added to a point, it can either be added as the first snippet for that point (`prepend`) or the last (`append`). Order is important since snippet execution can change the value of instrumentation variables.

In addition to calls to the basic primitives, instrumenta-

2. `$procedures` is a predefined variable that lists all procedures in the application program.

tion code blocks also can contain conditional expressions, calls to subroutines in the application, and references to variables in the application program. The following example uses an application variable (`timeSteps`) to start a timer after the 300th time step of the application.

```
(* if (readSymbol("timeSteps")==300) {
        startProcessTimer(x);
    } *)
```

Program instrumentation generally needs to be inserted into specific procedures in an application or library. For example, to compute metrics about a message passing library requires instrumentation to be inserted into the message passing functions. Often, the same instrumentation needs to be inserted in several procedures (if a library has several entry points). To make this operation easier, MDL includes a way to create lists of procedures. For example, a simple list declaration might be:

```
list pvm_msg_func is procedure {
    flavor pvm;
    items { "pvm_send", "pvm_recv" };
}
```

The first line defines the name of the list, `pvm_msg_func`, and the type of the list, `procedure`. The second line indicates that the list applies to PVM programs. The third line enumerates the procedures in this list. When the list is used during MDL evaluation, the listed procedures are looked up in the application program and any that are not present in the program are removed from the list.

2.2 Constraints

A key feature of MDL is its ability to constrain a metric description to different program components. Program components are identified using a hierarchical naming system. Each program component (e.g., procedure, file, process, etc.) has a unique name. Collections of program components are grouped together in resource hierarchies. For example, procedures are grouped together into modules, and modules are grouped together into a resource called Code. Therefore the procedure `func1`, located in file `test.c`, would be named `/Code/test.c/func1`.

To gather performance data about a specific program component, the metric description language provides constraint clauses. Constraint clauses create Boolean variables that are true when a specific program component is active. For example, a constraint on a module would be true whenever a procedure in that module is the currently executing procedure. The following is an example of a simple constraint clause that is true when a selected module is active.

```
constraint module /Code is counter {
    foreach func in $constraint[1].funcs {
        append preInsn func.entry
         (* module = 1; *)
        prepend preInsn func.return
         (* module = 0; *)
    }
}
```

The first part of a constraint clause describes the constraint name, `module`, and the resource hierarchy the constraint clause applies to (`/Code`). Within a constraint clause, the variable `$constraint` is bound to the name of the resource that has been selected. In this case if the metric user requested a metric restricted to a specific module, the resource named might be `/Code/foo.c`, and the `$constraint[1]` variable would be bound to `foo.c`.[3] Associated with the module described by `$constraint[1]` is an attribute (`funcs`) whose value is a list of the procedures defined in that module. The constraint shown in the above code fragment creates a Boolean variable that is true when a selected module is active.

The power of constraints is that they can be combined. For example, a module constraint could be combined with a message type (tag) constraint to restrict a metric to only those messages of a specific type that were sent or received by a single module. By combining constraints it is possible for metric users to request detailed metrics about application programs. Since each constraint corresponds to a Boolean variable, constraints are combined using an “and” operation.

2.3 A Complete MDL Definition

A complete metric definition contains a description of the code to insert and to compute the un-constrained metric (i.e., a metric computed for the entire application); it also a list of attributes about the metric, and a list of constraints that can be applied to the metric. Figure 2 shows a complete metric description. The first section of the definition includes information that is used to display the metric such as its name, and the units (operations per second). The `aggregateOperator` describes how the metric can be combined from different processes or threads to compute a single value for all threads of execution. Most metrics are combined using a sum operation, but minimum and maximum operators are also provided. The `flavor` field lists the different programming models where the metric is valid. Since not all metrics apply to all possible configurations this provides control over which metrics can be requested. The second part of the metric description lists constraints

3. The subscripts for resource names start at zero with the most specific (trailing component) of a name.

that can be applied to this metric. Since not all constraints may be appropriate to all metrics, only those that apply to the metric being defined are listed. The final section of the metric definition is the base clause that describes the instrumentation code necessary to compute the value of the metric for the entire program. In the example shown, the base metric increments a counter every time a function in the list pvm_msg_func is called.

```
list pvm_msg_func is procedure {
   flavor pvm;
   items { "pvm_send", "pvm_recv" };
}
constraint procedure /Code is counter {
   append preInsn $constraint[0].entry
       (* procedure = 1; *)
   prepend preInsn $constraint[0].return
       (* procedure = 0; *)
   }
}
metric msgs {
   name "Messages";
   units opsPerSecond;
   aggregateOperator sum;
   flavor { pvm };

   // Constraints that can be applied
   constraint module;
   constraint procedure;
   constraint msgTag;

   // the base computation of the metric.
   base is counter {
    foreach func in pvm_msg_func
      append preInsn func.entry constrained
         (* msgs++; *)
   }
}
```

Figure 2: A Complete Metric Description
Metric "msgs" counts messages sent by PVM message passing routines (listed in* `pvm_msg_func`*).

During application execution, a user (or higher level software) can request that a metric defined in MDL be enabled for a specific combination of program resources. For example, if the user requested that the msgs metric be enabled for the procedure /Code/myprog.c/foo, the code shown in Figure 3 would be inserted into the program. To satisfy this request, four instrumentation code snippets are inserted. The first two are inserted into the procedure foo. The one at the entry point to foo sets a counter to 1 when foo is called, and the instrumentation at the end of foo clears the counter when foo returns. These two snippets are inserted by the evaluating the constraint clause. The third and fourth code snippets are inserted into the message passing routines pvm_send and pvm_recv. These two statements conditionally increment a variable that counts the number of messages sent or received. The last two snippets are inserted by evaluating the base clause for msgs for each of the procedures defined in the list pvm_msg_func. Had the metric been requested for the entire program (i.e., un-constrained), snippets without if statements would be inserted at the entry to pvm_send and pvm_recv.

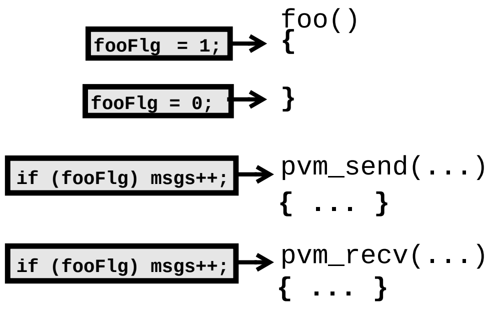

Figure 3: Instrumentation Generated for msgs metric constrained to the foo function.

2.4 Capabilities and Limitations of MDL

MDL has a simple type system consisting of two base types that can be used in instrumentation code and four types used to define where instrumentation code should be inserted. The instrumentation code types, counter and timer, can be used in instrumentation code as integer variables and to record the time between events respectively. The four types used to define where instrumentation can be inserted are: procedure, module, list, and iterator. Procedure is an aggregate type that describes a subroutine and contains fields for the entry point, return statements(s), and subroutines called. Modules are collections of procedures. List is an aggregate type that represents a collection of variables of the same type. Lists may be accessed sequentially using iterators or randomly using array subscript notation (square brackets).

MDL can be used to describe many types of metrics. However, since the instrumentation code lacks a looping construct, the language is not Turing complete. We choose not to include a looping construct since it would make it impossible to predict the execution time of the instrumentation code. With MDL it is currently possible, given the execution time of called application subroutines, to develop a fairly accurate model of the cost to execute each instrumentation snippet to be inserted. In Paradyn, we use this cost information to control the amount of instrumentation inserted into the program [6]. For the metrics we have written to date, we have not found the absence of a looping statement a limitation to expressing any metrics we have wanted to create. As we gain more experience with MDL, we will evaluate whether the absence of a looping construct unduly limits the type of metrics that can be created

3 Structural Analysis: Parsing the Binary File

Paradyn performs a simple form of structural analysis to identify instrumentation points in the application program. The result of this analysis is a list of the instrumentation points for each function, annotated with the address of the point, the original instruction at the point, and additional instructions that can be replaced when the point is instrumented. Other useful information about functions is also obtained during the analysis of the executable. We can determine if a function is a leaf function, and if the function creates a new stack frame. This information can be useful later, when we do a stack trace for inserting or deleting instrumentation as described in Section 4.

Paradyn finds the points by analyzing the executable file(s) of the application. Our goal is to be able to handle an arbitrary executable file, gathering information from both the symbol table and by scanning the binary image. When we identify a point, we also determine how instrumentation could be inserted at that point. Our analysis uses many of the techniques used in binary rewriting [11].

An executable file is processed in several steps. In the first step we process the symbol table to get the size and address of the code and data segments. The result of this step is a platform-independent representation of the executable file consisting of pointers to the code and text segment (in the memory mapped executable file), and a list of symbols (functions and data objects) annotated with name, type, starting address, size, and (in some cases) the module to which each symbol belongs. An important piece of information, not directly available on all platforms, is the size of each function. The size is needed when we look for instrumentation points. If the size is not in the symbol table, we infer the size by locating the starting address of the next function or data object. This inference step is needed even on platforms that supposedly provide the size directly, since we have found that many functions do not have the correct size in the symbol table. On the AIX/Power2 platform, we determine the end of a function by scanning for the signature of a trace-back record that follows each procedure in the extended COFF file format. On some platforms (Solaris and HP-UX), we read symbolic debugging information to find the module (source file) of the functions, since this information cannot be derived from the standard symbol table.

Once we have the list of functions, the second step is to find the instrumentation points for each function. The instrumentation points currently provided are function entry and exit points, and call sites. The entry point for each function is defined as the starting address obtained from the symbol table. The basic method for finding the other instrumentation points is to sequentially scan the code of each function, beginning from the entry point, searching for instruction sequences that implement calls or exit. This step is platform dependent. Scanning the instructions is trivial in most RISC platforms, as all instructions are of the same size; we perform a simple matching. On the x86, where instructions have different sizes, we decode each instruction to determine its length before processing it, so that we can find the start of the next instruction.

In many cases, we can find function exit points and call sites by looking for return and call instructions. Any jumps that leave a function are also defined as exit points. Indirect jumps are more difficult to analyze, since we do not know the target at analysis time, and therefore do not know if an indirect jump is leaving a function. The basic approach is to assume that indirect jumps do not leave a function, except that in some cases we can use heuristics to find the target of the indirect jump. For example, SPARC code uses an idiom where the target of a jump is loaded into a register, and then an indirect jump through that register is used:

```
sethi HI(addr), reg
or LOW(addr), reg, reg
jump reg
```

This code sequence is frequently used for jumps where the distance between the target and the jump instruction is greater than the value that fits in the immediate field of the jump instruction. If we find this instruction sequence in the code and the target is outside the function, then the jump will be defined as an exit point. In cases where this is not possible, we can insert code to dynamically check the jump destination address.

In addition to finding instrumentation points, the analysis of the executable also finds additional instructions, either before or after a point, that can be used when instrumenting the point. These additional instructions are needed in cases where we replace several instructions to insert a jump at that point (several instructions are needed to build jumps with a distant destination). We check that there are no jumps into the middle of these instruction sequences, so we can collect the targets of all direct jumps as we scan the instructions looking for instrumentation points.

Several compilers put data in the code segment in a way that can make it difficult to distinguish between data and instructions when we scan the executable file. The most common data that is mixed with code is a jump table for switch statements. We use heuristics that find and skip jump tables, and are effective on the code generated by the GNU, Sun, and Microsoft Visual compilers. For example, many indirect jumps are of the form:

```
jmp dword ptr[reg*4+addr]
```

where `reg` is a general register, and `addr` an immediate value that gives the base address of the jump table. Depending on the base address, we can tell if the jump table is within the code segment or not. Although we do not know the size of the table, we can infer it by looking at the addresses in the table. As long as we find an address that is within the current function, we assume that it is part of the jump table.

4 Inserting and Deleting Instrumentation

A major challenge of dynamic (run time) instrumentation, that is not encountered in binary rewriting, is interacting with the execution state of the application program. To instrument a program at run time, we generate instrumentation code fragments and place them in dynamically allocated patch areas called *trampolines*. Once code is generated and placed in the trampolines, it must be tied into the application program. The Instrumentation Manager is responsible for the insertion of instrumentation into and deletion from the application process. The complete details for trampolines are described in Section 5.

Each block of generated instrumentation code must be tied to an instrumentation point in the application processes. To insert this code, we stop the application process and install the code and data (counters and timers) into the application address space using operating system facilities (for UNIX systems, ptrace or the /proc file system). After the code is generated for the trampolines, the original instruction(s) at the point are relocated to the trampoline. The original instruction(s) at the instrumentation point are then modified to jump to the base trampoline. When the instrumentation is disabled, we first remove the branch into it from the trampoline, and them reclaim its memory when the code is no longer active.

4.1 Instrumentation Insertion

Inserting instrumentation requires creating the trampolines (described in Section 5.1) and modifying the original instructions in the instrumentation point to jump to the base trampoline. If there is some instrumentation already at that instrumentation point, we use the existing trampoline.

When instrumentation is inserted at a point, the original instruction at the point must be changed to a jump to the trampoline. Sometimes, the instruction at the point can be replaced with a single jump instruction. However, in general we may have to replace multiple instructions at the point, either because the size of the instruction at the point is smaller than the size of a jump instruction, or because we need several instructions for a jump. First, we must make sure that there are no jumps into some location in the middle of the sequence of instructions being modified. We handle this case by checking that no direct jump in the application jumps into the middle of an instruction sequence that we want to modify. If there is any jump into the middle of an instruction sequence, we cannot modify that sequence. If we can not, there are several alternatives: insert a trap instruction (which requires the modification of only one instruction, and always can be safely inserted); copy the larger instruction sequence (including the jump) to the trampoline, padding the code to make room for instrumentation code and modifying the jump; or reject the instrumentation request for that point.

Checking the targets of all direct jumps is not enough, however, since there may be indirect jumps where the target cannot be determined until execution time. In those cases we use heuristics to find the targets of indirect jumps statically, and this works well for all of the code that we have encountered from the GNU and commercial compilers. As we increase the abilities of our structure analysis to include full control and data dependences, this becomes less of an issue. The general solution to this problem is to instrument those indirect jumps that cannot be analyzed and check the target at run-time, taking some special action if we find that the target is in the middle of a modified instruction sequence, but we have not implemented this solution yet as we currently have no demand for it.

Sometimes we want to instrument functions that are too small to insert jumps to a base trampoline at the entry and exit points. This often happens with C library functions such as read and write. In this case, we relocate the entire function to a different location, near the base trampoline, and use short jumps, which in most platforms require the modification of only one instruction. The original function is then replaced with a jump to the relocated function.

Another problem with modifying multiple instructions is that the program might be executing that particular instruction sequence at the moment we stop it to insert instrumentation. If we simply resume the program after modifying that instruction sequence, it will begin executing in the middle of the new code, which would cause unpredictable results. A simple solution is to modify the application's PC so that the code resumes in the relocated instructions in the trampoline (instead of in their original location). We can also insert a breakpoint (trap) at the end of the sequence and run the program and wait until it reaches the breakpoint.

4.2 Instrumentation Deletion

When a delete request is made, we prevent future activations of the instrumentation by modifying the jump to the trampoline to bypass it. The memory block that contains the deleted code can also be freed, but first we must make sure that the code is not active. Requests to delete instrumentation code are collected and handled in a batch to amortize the fixed overhead involved in deletion. The queued deletes are processed when the list exceeds a predefined length or when the total free space drops below a specified threshold.

To delete instrumentation, we first stop the application process and examine the PC and call stack. We check the PC and current stack frames to make sure that the instrumentation code is not currently being executed. If it is currently being executed, we defer the deletion, placing the deletion request back on the list and this deletion is attempted when deletion is next triggered.

When we delete the instrumentation code, we must also delete the data (counters and timers) used by that code. Since this data is shared by multiple instrumentation points, we use a reference count for each counter/timer to control deallocation.

We also handle fragmentation of the instrumentation heap by compacting the heap and updating all branch destinations and data addresses. The compaction algorithm is run when a request cannot be satisfied from the free list.

4.3 Dealing with Compiler Optimizations

A major challenge for run-time tools is coping with compiler optimizations. Users are interested in improving the performance of their applications, and must be able to measure their programs with compiler optimizations enabled. Many optimizations will not affect our ability to instrument code and obtain meaningful performance results, but certain optimizations could lead our instrumentation to produce incorrect results. Although we do not have a general method for dealing with compiler optimizations, we can handle specific optimizations in such a way that we can get meaningful results. One of these techniques is to undo compiler optimizations that fuse multiple instrumentation points together when an instrumentation request for a fused point is made.

Some compilers use a *tail-call optimization* for functions that make a procedure call as its last operation. The optimized function will not return to its caller; instead it gives its return address to the called function, which then returns directly to the caller. On the SPARC platform, the tail-call optimization is usually implemented by having the instruction in the delay slot of the call instruction modify the register where the return address of the call is written. For example:

```
call f
restore
```

This kind of optimization can make it difficult to collect correct performance data. If we treat the call as a normal call site, we miss the exit point of the function. An alternative is to treat the tail call as an exit point, but then we could miss the time spent in the called function. Our approach to this problem is to undo the tail-call optimization so to gather accurate performance data. Whenever we instrument such a point, instead of relocating the instructions, we generate new code in the trampoline that emulates the original code, but without the optimized tail-call. For the code fragment above, we generate the following code:

```
restore
st      %o7,[%fp+0x44]
call    f
nop
retl
ld      [%fp+0x44],%o7
```

This de-optimization is used most commonly when instrumenting C library functions (such as “read”).

5 Compiling and Code Generation for MDL

Compiling and generating code for MDL takes place at two points in time (see Figure 1). The first stage occurs when Paradyn (“MDL Parse”) processes its configuration file containing the basic metric definitions; these definitions are compiled into parse trees. The second, and more interesting stage (“Code Generator”), occurs when a request is made to instrument the running program. Machine code is then generated to satisfy the request. Code generation is an incremental process, with each new request for instrumentation generating a new fragment of code in the application program.

In this section, we first present the basic structure of the instrumentation code. We then describe the code generation for the instrumentation primitives and predicates. Next, we describe optimizations we use at various points in the code generation. All of the features described in this section are available on all supported architectures: Sun SPARC, HP PA-RISC, IBM Power2, DEC Alpha, and Intel Pentium architectures, accommodating the idiosyncrasies of the various architectures.

5.1 Trampolines: Tying It Together

The first step for inserting code into an application is to allocate space for the dynamically generated code. Code generation is done incrementally, as each new request for instrumentation is made. At a given point in the application program, instrumentation code can be inserted or deleted. We use small code fragments, called trampolines, as the mechanism to tie this all together. Associated with each active instrumentation point is a *base-trampoline*, and each block of instrumentation code is placed in its own *mini-trampoline*. The base trampoline contains the relocated original instructions from the instrumentation point in the application program, instructions to save and restore registers, slots where jumps to mini-trampolines can be inserted, and a jump to return to the application code. The mini-trampolines contain instrumentation code followed by a jump. If there are multiple instrumentation requests for a point, the mini-trampolines are chained together (much as is done in Synthesis [12]), with the last trampoline in the list jumping back to the base trampoline. To trigger the instrumentation code, instructions are inserted at the point in the application code to jump to the base trampoline (see Section 4.1). The cost of the instrumentation at a particular instrumentation point is updated in the base trampoline (for both pre-instrumentation and post-instrumentation). In this way, Paradyn can keep track of how much perturbation has been introduced into the application. Figure 4 shows the structure of base and mini-trampolines.

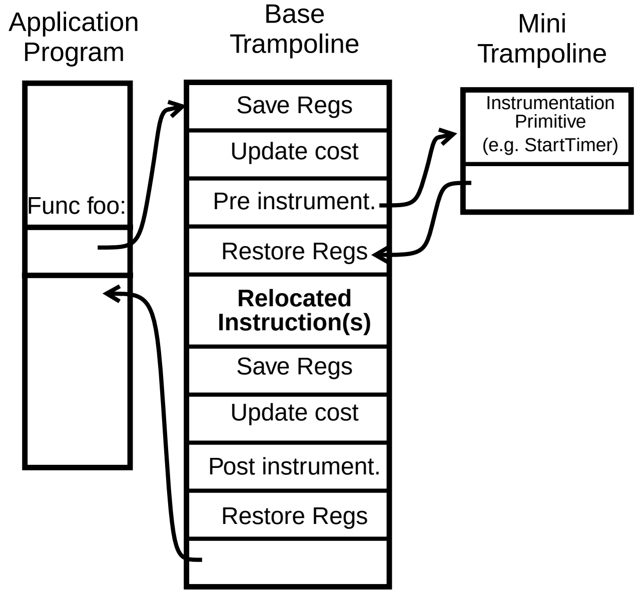

Figure 4: Application Code, Base and Mini-Trampolines

For the base trampoline, we need to reserve space to hold the relocated instructions from the application. Normally, we relocate two or three instructions (two for a `call` and its delay slot and some times one additional `save` before the `call`), but we might need to relocate extra instructions. For example, on the SPARC if there is a function that returns a structure, then the instruction after the delay slot has the size of the structure and it has to be relocated too. One `nop` is added when relocating a jump instruction, since we currently do not try to fill delay slots in relocated instructions.

The mini-trampoline in Figure 5 illustrates an example of an increment primitive (similar to what would be generated at the entry to `foo` in Figure 3): it loads the value of the counter, increments the value of the counter, stores the value of the counter and then jumps back to either the base trampoline or to the next mini-trampoline.

When we relocate a jump or branch instruction to the base trampoline, we need to modify that jump, since in

```
// Instrumentation code (incr primitive)
//       Load counter
minitramp:          sethi  %hi(0x61800),%l0
minitramp+4:        ld  [%l0+0x3e0],%l0
//       Increment counter
minitramp+8:        inc  %l0
//       Store counter
minitramp+12:       sethi  %hi(0x61800),%l1
minitramp+16:       st  %l0, [%l1+0x3e0]
// Branch to either base trampoline or next
// mini-trampoline
minitramp+20:       b,a   basetramp+44
minitramp+24:       nop
```

Figure 5: Mini-trampoline (SPARC architecture)

most architectures the range of jump instructions is limited. We may have to generate multiple instructions to emulate the original jump, since the destination can be further away than the displacement of the original jump instruction.

For the x86 architecture there is an additional difference in the base trampoline. Since instruction length is variable, we may need to relocate up to five instructions, the instruction at the point, plus additional instructions before or after the point. These extra instructions are moved to make sufficient space for the jump to the base trampoline. Additional instructions that are taken from before the instrumentation point are relocated to the beginning of the base trampoline, and additional instructions from after the point are relocated to the end of the base trampoline, just before the return to the application code. This permits us to include a jump instruction in the relocated sequence:

```
L1:
    ...
    je L1
    leave
    ret
```

The `ret` instruction is an instrumentation point, and since the instruction is only one byte, we need to relocate additional instructions. The `je` instruction can be relocated as long as there is no jump to the `leave` or `ret` instructions. Since the `je L1` is relocated to the beginning of the base trampoline, no instrumentation will be executed if the jump is taken:

```
Base trampoline:
    je L1
    leave
    <pre point instrumentation>
    ret
    <post instrumentation> (not reached)
    <return to user code>
```

Instrumentation of a function return point is handled in a special way on the Power platform. Because it can be difficult to determine precisely the exact return points, we use the following approach. We insert instrumentation at the entry point to replace the value of the link register, which contains the return address, with the address of the base-trampoline for the return point. The original return address is saved on the stack so that it can be restored later. When the function returns, it will jump to the base-trampoline, which executes the instrumentation code, retrieves the real return address from the stack, and jumps to that location.

5.2 Optimizations

There are several places where optimizations are made to our dynamic code generation. Optimization in our environment has several unique aspects because we are interacting with both existing application code and with previously inserted instrumentation code. To avoid repeated address computation, we perform common sub-expression elimination. To avoid long streams of redundant instrumentation when collecting data on many instances of a resource, we employ a simple form of vectorization.

A simple optimization is eliminating common sub-expressions. This type of operation is particularly effective, since we frequently have multiple references to computed addresses. During code generation, we use the standard techniques of generating the code once for such an expression and keep the result (if possible) in a register until the last occurrence of the sub-expression.

Collecting performance data for many instances of the same resource can generate repetitious and expensive instrumentation. For example, this situation can occur when profiling many memory blocks. In each case, there could be hundreds or thousands of instances of each resource. Since each instrumentation request is generated separately, we need to detect independent requests and vectorize them. Without vectorization, the generated code might look like:

```
if (<other constraint>)
    if (arg[1] == instance1)
        counter1++;
if (<other constraint>)
    if (arg[1] == instance2)
         counter2++;
. . .
if (<other constraint>)
    if (arg[1] == instanceN)
        counterN++;
```

The first `if` statement represents a constraint not related to the vectorization (such as constraining to a particular procedure). The second `if` statement in each group is the target of the vectorization optimization. Note that this optimization is similar to that used in a “switch” statement. The `arg[1]` term specifies the resource instance; in this example, it means that the instance is identified by parameter 1 of the function in which this instrumentation was inserted. The number of instructions inserted is linearly proportional to the number of resource instances. With this amount of instrumentation, the perturbation could be intolerable.

Instead, we can allocate a vector of counters/timers for all resource instances, and insert code to index into the appropriate cells of the vector. The optimized code looks like:

```
if (<other constraint>) {
    index = f(arg[1],resource);
    vector[index]++;
}
```

In this example, `f()` is a user-supplied function that maps from the resource instance to the vector index. This function also takes as input a description of the resource class (`arg[1]` in this case). The number of instructions inserted is a constant independent of the number of resource instances. This optimization is especially effective when the mapping function can produce a dense vector. To further improve this optimization, we pre-calculate and propagate all constants at the time of instrumentation.

As an example, we have measured operations (cache misses) for each memory block (cache line) in arrays in the application program. We allocate a vector of counters, one counter associated with each memory block the array. The resource in this case is the user array, and the information that we use in the mapping function is the base address and length. The resource instance specified by `arg[1]` is the memory address. Function `f()` maps `arg[1]` to a cell in the counter vector based on the array information and the memory block size. For this example, the unoptimized instrumentation executes $N(X+15)$ instructions in the mini-trampoline, where N is the number of resource instances and X is the number of instructions for register save/restore and other constraints. With vectorization, the number of instructions is X+28. A peephole optimization could further reduce the number of instruction to $X+20$. Since memory objects are numerous, vectorization saves a substantial amount of time.

6 Performance Measurements

The critical performance cost for our instrumentation is the time it takes to go from the user code to the trampolines and back. The cost of the instrumentation code itself (the part that calculates the metrics) is specific to the semantics of the metric description, so it is difficult to make general statements about it. For the counter and timer operations, we have previously reported these costs [7].

The results in Figures 6 and 7 show the performance of our code patching system on four different processor architectures. The overall time shown in the fourth row of Figure 6 reflects the time required to execute instrumentation to call an empty procedure at a single point in a program. Due to our need to preserve machine state and

	x86, 200MHz, Pentium-Pro	SPARC, 110MHz, microSPARC II	Power, 66MHz, Power2	PA-RISC, 50MHz, HP 9000/715
Call to empty procedure	40ns	87ns	156ns	261ns
+ base trampoline w/jump (no saves/restores)	67ns (1.7)	209ns (2.4)	226ns (1.4)	633ns (2.4)
+ base trampoline w/trap (no saves/restores)	42,120ns	n/a	n/a	n/a
+ base trampoline w/jump (with saves/restores)	264ns (6.6)	504ns (5.8)	365ns (2.3)	1374 (5.3)
+ empty mini-trampoline	281ns (7.0)	556ns (6.4)	465ns (3.0)	1632ns (6.3)

Figure 6: Cost of Triggering Instrumentation.

The 1st line is the cost of a procedure call with no parameters, with no instrumentation in place. The 2nd line adds the jump to the base trampoline and back, with no register save/restores and no jump to mini-trampolines. The 3rd line uses a trap instruction instead of a jump to get to the base tramp. The 4th line adds register save/restores (to the jump case). The 5th line adds to the base tramp the register save/restores and a jump to an empty mini-trampoline. Numbers in parentheses are cost relative to the empty procedure call.

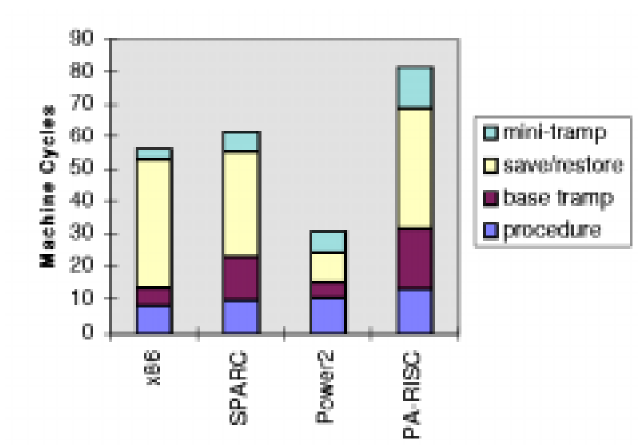

Figure 7: Cost of Operations Normalized by Clock Speed

squeeze the code into an existing binary image, the times are somewhat higher than a traditional compiler would generate for a procedure call.

For most architectures, the largest share of the time is spent saving and restoring machine state. For example, on the x86 platform, 70% of the time is spent saving and restoring machine state. Most of this time is spent preserving one register, *flags*. Saving and restoring this one register required 112ns or 40% of the total instrumentation time. On the SPARC, we use the register wheel to preserve the local and parameter registers, but must still preserve the general and floating point registers to ensure correct behavior. The save and restore time is significantly shorter on the Power2 architecture, since we only save registers when they are used by the resulting instrumentation. A more thorough analysis of the instrumentation code could be used to decide if we really need to save registers at a specific instrumentation point.

The third row of Figure 6 also shows the time required to execute a null procedure when we have to use a trap instruction. As expected, since this requires a trap into the operating system kernel, this time is significantly longer than simply executing a branch instruction (42us versus 281ns). However for many uses such as data breakpoints, this approach is much faster than a trap instruction and a context switch to a separate debugger process.

There are also several other opportunities to improve the performance of our instrumentation system. First, the base trampoline is a static code template with place holders (nop's) for all possible situations. By customizing the base trampoline for each situation, we could improve the time of base trampoline execution for most cases. Likewise, we could fuse the base and mini-trampolines to reduce the number of branches by two. This change would improve run-time performance at the cost of greater complexity in managing multiple mini-trampolines at a single point.

7 Conclusions

In this paper we have introduced MDL, a language for dynamic program instrumentation. MDL provides a machine independent way to describe the instrumentation to be inserted into an application program. The language also includes features to de-couple the specification of what data to collect from how to constrain data collection to particular program components such as procedures. In addition, MDL permits customization of data collection to different parallel programming models by permitting metric definitions to be tagged for specific programming models.

The second part of this paper described the mechanisms we used to create a run-time instrumentation compiler. Our run-time code instrumentation system differs from binary editors and run-time code specialization tools since we need to both generate code at run-time *and* weave it into arbitrary points in a binary image during its execution. We presented techniques to squeeze instrumentation into tight code sequences, including moving code and selective de-optimization of compiler generated code. We also reported ways to insert and delete code from a running program, and optimization techniques for run-time code generation.

We allow instrumentation of optimized code on all our supported platforms, including the commercial compilers such as IBM xlc, Microsoft Visual, and Sun Sparcworks. As compiler technology advances, our dynamic instrumentation will have to advance with it. Our ability to handle optimizations in the future may depend on the compiler providing more information about the nature of its transformations.

Our run-time code generation system currently produces code for the SUN SPARC, HP PA-RISC, DEC Alpha, IBM Power2, and Intel x86 architectures. All features described in this paper have been incorporated into the Paradyn parallel performance tools. We have also created an API to export the run-time compiler features for other uses.

References

[1] A. Beguelin, J. Dongarra, A. Geist, and V. S. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *IEEE Computer (***26**) 6, June 1993.

[2] I. Nashon and D. Berstein. FDPR: A Post-pass Object-code Optimization Tool. *International Conference on Compiler Construction*, Linkoping, Spring Verlag LNCS (April 1996).

[3] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. *SIGPLAN PLDI Conf.*, San Francisco (June 1992).

[4] M.T. Heath and J.A. Etheridge. Visualizing Performance of Parallel Programs. *IEEE Software* (**8**) 5, Sept. 1991.

[5] J.K. Hollingsworth, R.B. Irvin, and B.P. Miller. The Integration of Application and System Based Metrics in a Parallel Program Performance Tool. *3rd ACM Symp. on Principles and Practice of Parallel Programming*, Williamsburg, VA (April 1991).

[6] J.K Hollingsworth and B.P. Miller. An Adaptive Cost Model for Parallel Program Instrumentation. *Euro-Par '96*, Lyon, France (August 1996).

[7] J.K. Hollingsworth, B.P. Miller, and J.M. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. *Scalable High Performance Computing Conference*, Knoxville, TN (May 1994).

[8] H. A. A. Hough and J. E. Cuny. Perspective Views: A Technique for Enhancing Parallel Program Visualization. *International Conference on Parallel Processing,* Vol.II (August 1990).

[9] J.K. Hollingsworth and B. Buck. DyninstAPI Programmer's Guide. University of Maryland *Computer Sciences Technical Report* CS-TR-3821, (August 1997).

[10] D. Kimelman. Environments for Visualization of Program Execution. In **Performance Instrumentation and Visualization**, M. Simmons and R. Koskela, editors. Addison-Wesley, 1990.

[11] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA (June 1995).

[12] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel., *12th Symposium on Operating Systems Principles*, Operating Systems Review (**23)** 5, December 1989.

[13] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, (**28**)11, November 1995.

[14] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. *2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, (October 1996).

[15] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. *15th Symposium on Operating Systems Principles*, Copper Mountain, CO (December 1995).

[16] R. Rastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. *Winter Usenix Conference* (January 1992).

[17] 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. *Scalable Parallel Libraries Conference*, A. Skjellum, Editor. 1993, IEEE Computer Society.

[18] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, and O. Wolf. Event and State-Based Debugging in TAU: A Prototype. *SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, PA (May 1996),

[19] A. Srivastava and A. Eustace. ATOM: A system for Building Customized Program Analysis Tools. *SIGPLAN Conference on Programming Language Design and Implementation,* Orlando, FL (1994).

[20] B. Topol, J. T. Stasko and V. S. Sunderam. Monitoring and Visualization in Cluster Environments. *Technical Report GIT-CC-96-10*, Georgia Institute of Technology (March 1996).

[21] J. C. Yan, S. R. Sarukkai, and P. Mehra, "Performance Measurement. Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit. *Software Practice & Experience,* (**25)** 4.