Paradyn Parallel Performance Tools

MDL Programmer's Guide

Release 5.1
May 2007

Paradyn Project
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
paradyn@cs.wisc.edu

Table of Contents

1 INTRODUCTION

Paradyn is a parallel performance measurement tool that performs *dynamic instrumentation* on running applications. Instrumentation is a sequence of instructions, such as a code fragment inserted at the beginning of a function to increment a count of the number of times the function is invoked. The instructions are compiled from a source language, called the *Metric Description Language* (MDL). For Paradyn to properly instrument (and thus measure) your application's performance, you must specify all the metrics you will need using MDL in a *Paradyn configuration file*. Commonly used metrics have already been programmed for you as part of the Paradyn distribution in a file named `paradyn.rc`.

This paper is written to help you become more proficient in programming with MDL. Specifically, this paper is designed to save you time and frustrations in using this new language. MDL is unlike any language that you might have used, so we want to ease your introduction. We assume you have already familiarized yourself with the MDL language syntax: if not, please refer to the ***Paradyn User's Guide***. We further assume you are proficient in programming; MDL is an unusual programming language, as you will shortly discover, and understanding advanced programming concepts will ease your learning.

This paper walks you through from the bottom up. We first start with the basic MDL metric data types: counters and timers. The rest of the paper essentially goes through tips on how to properly use them. Even seasoned programmers will be surprised by how subtle the programming issues are when using even seemingly simple concepts such as counters and timers. Since the counters and timers do not update themselves automatically, we go on to actual instrumentation code that updates them.

Three types of instrumentation can be specified in MDL, with each type named by their location (or *point* of insertion): entry-point, return-point, and callsite. Entry-point instrumentation is inserted at the beginning of functions. Return-point instrumentation is inserted at the end of functions. Callsite instrumentation is inserted at places where functions call other functions when dealing with exclusive metrics. Entry-point and return-point instrumentation are closely related, so we consider them together in one section. Callsite instrumentation is described in a separate section.

Last, we devote a section on miscellaneous issues that did not fit nicely into any of the above mentioned sections and did not warrant separate sections themselves.

To simplify the presentation, we will be providing only MDL code fragments, and not complete metric implementations. In addition, though *constraints* are key to instrumentation, none of the MDL examples are `constrained`. That is, all examples will be written as if for whole-program metrics. Again, this is done to simplify the text. The metrics you write, however, will usually need to be `constrained`.

2 COUNTERS

Counters are one of two metric data types supported by MDL. The second, timers, will be covered in Section 3.

2.1 Counters are initialized to zero

Therefore, if you are using a counter as a flag, try NOT to use zero as the “okay” value.

Here is a fragment of a metric that counts the number of I/O function calls, *except* those calls made from within MPI functions. (MPI is a library of functions used for exchanging values in parallel programs.)

```
// Good example of using a counter as a flag

counter not_in_mpi;  // Flag is 1 when program is outside an MPI function

foreach func in mpi_funcs {
  append  preInsn func.entry  (* not_in_mpi = 0; *)
  prepend preInsn func.return (* not_in_mpi = 1; *)
}

foreach func in io_funcs {
  append preInsn func.entry (* if (not_in_mpi == 1) io_ops++; *)
}
```

The `not_in_mpi` flag is set only after a return from an MPI function, and we increment the `io_ops` counter only if the flag is on. That is, we are incrementing the counter of I/O function calls only when we are sure we are not executing within an MPI function. Assuming that the MPI functions do not call each other, directly or indirectly, it is impossible for the `not_in_mpi` flag to have a value of 1 when within an MPI function. To show this, we need a case analysis. At the moment of instrumentation, the program is either executing an MPI function or it is not, so there are only two cases to consider. (Note that if a program is currently within a function at the moment of instrumentation, any new entry-point instrumentation of the function may not be executed for the current invocation of the function.)

Case 1: Program is inside an MPI function at the moment of instrumentation.
Since `not_in_mpi` is initialized to zero, the flag has the correct off value. As a result, we will not mistakenly increment the I/O function calls count when inside an MPI function.

Case 2: Program is outside an MPI function at the moment of instrumentation.
Since `not_in_mpi` is initialized to zero, the flag has the incorrect off value. However, this metric was written for an MPI application, so this situation will be corrected shortly at the next call of an MPI function.

Suppose we mistakenly used zero as the “okay” flag value for this metric, as in the following example.

```
// Bad example of using a counter as a flag

counter in_mpi;  // Flag is 1 when program is inside an MPI function

foreach func in mpi_funcs {
  append  preInsn func.entry  (* in_mpi = 1; *)
  prepend preInsn func.return (* in_mpi = 0; *)
```

```
}

foreach func in io_funcs {
  append preInsn func.entry (* if (in_mpi == 0) io_ops++; *)
}
```

Using a case analysis, we find that it is possible to mistakenly count I/O function calls made from within MPI functions.

Case 1: Program is inside an MPI function at the moment of instrumentation.
Since `in_mpi` is initialized to zero, the flag has the incorrect off value. As a result, we *might* mistakenly increment the I/O function calls count when inside the currently executing MPI function.

Case 2: Program is outside an MPI function at the moment of instrumentation.
Since `in_mpi` is initialized to zero, the flag has the correct off value.

You may notice that deciding on the "okay" value of a flag depends on the errors the metric must avoid. In the above example, it was required that I/O function calls by MPI functions not be counted, so we had to choose 1 as the "okay" value. However, there may be situations where you must use 0 as the "okay" value.

3 TIMERS

Timers are more complex than counters, but they still can be used with a few simple rules. Paradyn has four timer functions: `startProcessTimer()` and `stopProcessTimer()` for virtual clock timers, and `startWallTimer()` and `stopWallTimer()` for wall clock timers. For clarity, we shall sometimes use the abbreviations `startTimer()` and `stopTimer()`.

3.1 Every `startTimer()` must be matched by a `stopTimer()`

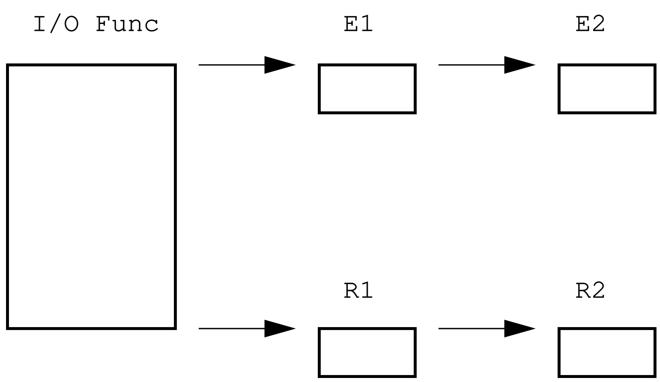
This is similar to the syntax of many programming languages. In Pascal, a `BEGIN` is ended by the first unmatched `END`. In C, an open brace `{` is ended by the first unmatched close brace `}`. In Lisp, an open parenthesis `(` is ended by the first unmatched close parenthesis `)`. In MDL, a `startTimer(T)`, for some timer `T`, is ended by the first unmatched `stopTimer(T)`. An unmatched `stopTimer()` has no effect. Calling `startTimer()` on an already running timer neither restarts the timer nor starts a new copy of the timer. However, the timer is actually stopped ONLY when every `startTimer()` has been matched by a `stopTimer()`.

Unlike the Pascal, C, and Lisp examples, MDL timer matching is done as the instrumentation is executed, and not when the metric was written by you. That is, unmatched elements in Pascal, C, and Lisp will be caught by a compiler, but unmatched MDL timers will result in incorrect timer values. At this point, we just want you to be aware of the above rule. The later sections on instrumentation will show you how to keep `startTimer()` and `stopTimer()` matched.

3.2 Use append/append instead of append/prepend for timers

Software instrumentation inherently perturbs the original program, so the program's timing behavior changes. Furthermore, subsequent instrumentation can change the timing behavior of previous instrumentation. The following example shows how relative errors in timers can be minimized by careful relative placement of timer instrumentation blocks. Let us illustrate this by examining instrumentation that times the execution of an I/O function.

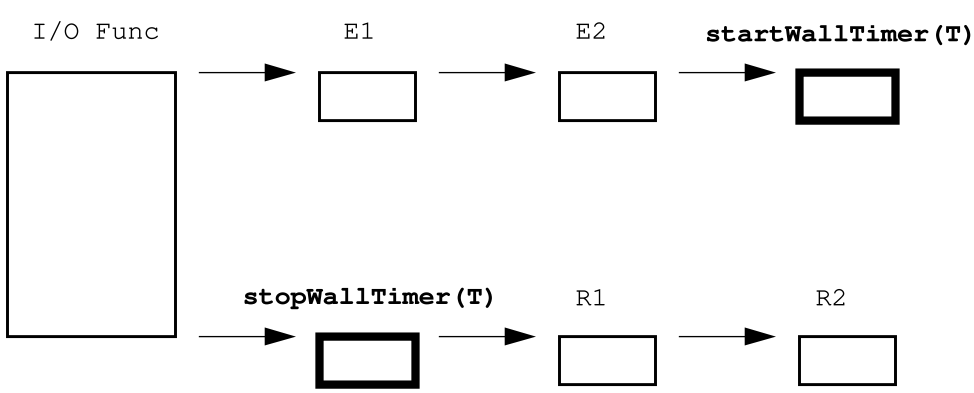
We start with an I/O function that has already been instrumented as follows. `E1` and `E2` are two pieces of entry-point instrumentation. `R1` and `R2` are two pieces of return-point instrumentation.



Now we want to insert our timer instrumentation, written in MDL as follows.

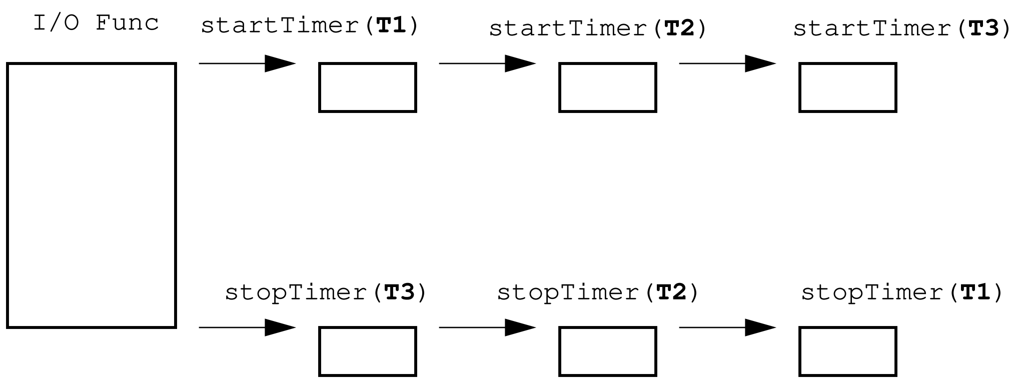
```
// The most obvious way to implement an I/O function timer metric

foreach func in io_funcs {
  append  preInsn func.entry  (* startWallTimer(T); *)
  prepend preInsn func.return (* stopWallTimer(T);  *)
}
```

The following figure shows how things would look after we have actually instrumented the I/O function with the code given above.



Notice that for a single timer metric, this is the best placement of the instrumentation. By `append`ing `startWallTimer(T)`, we avoid including the execution times of the existing entry-point instrumentation. By `prepend`ing `stopWallTimer(T)`, we avoid including the execution times of the existing return-point instrumentation. However, let us now consider what happens when multiple timers are inserted. For example, we may have inserted one timer to keep the I/O time for the entire program, one timer to keep the I/O time for a *module* (a set of procedures), and one timer to keep the I/O time for a particular procedure.

Below we have instrumentation for timers `T1`, `T2`, and `T3`; inserted, in that order. For the sake of clarity, let us assume there was no preexisting instrumentation on the I/O function.

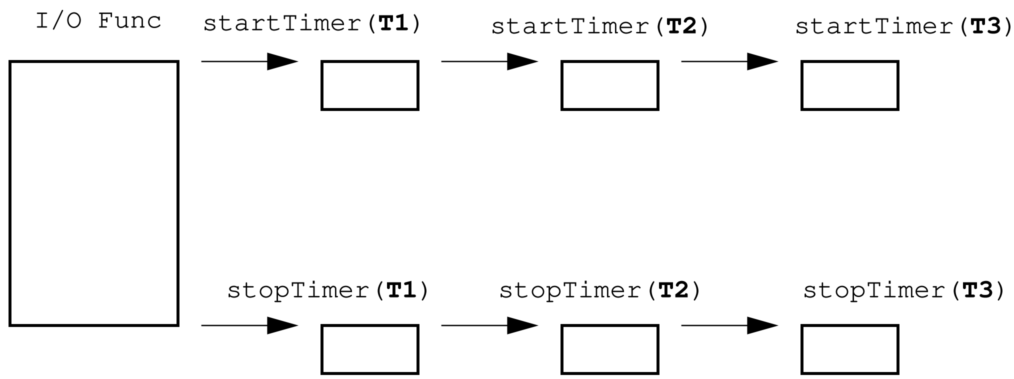


Notice that `T3` gets the best placement for being inserted last. Meanwhile, `T1` gets the worst placement after being inserted first. As new timers are inserted, the earlier timers get less and less accurate because they must also time *all* the new timers' instrumentation.

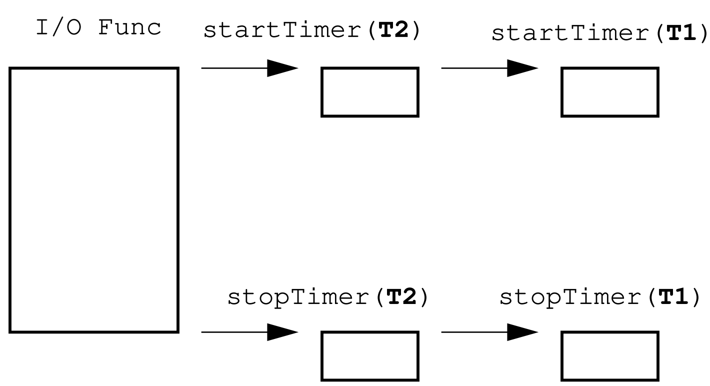
To reduce the rate of decrease in timer accuracies, we can use `append/append` instead of `append/prepend` in our metrics. That is, the MDL description becomes the following.

```
// An I/O function timer metric that causes less interference

foreach func in io_funcs {
  append preInsn func.entry  (* startWallTimer(T); *)
  append preInsn func.return (* stopWallTimer(T);  *)
}
```

Here again we have instrumentation for timers `T1`, `T2`, and `T3`; inserted, in that order. However, this time, the return-point instrumentation has been `append`ed, instead of `prepend`ed. As new timers are inserted, the earlier timers still get less accurate, but the rate of decrease in accuracy has been lessened because they must time only *half* the new timers' instrumentation. Also, notice that if all the instrumentation pieces take the same amount of time to execute, each timer will see the same amount of error. This may be better because then you won't have metrics disagreeing with each other about the amount of time this I/O function takes.



Using `prepend/prepend` also has the same relative-error minimization property, but reduces the chances of instrumentation being executed at the earliest possible opportunity. To see this, the following figure illustrates the situation after instrumentation for `T1` and `T2` have been inserted with `prepend/prepend`. Suppose the program is executing the `startTimer(T2)` instrumentation when we need to insert instrumentation for `T3`. Since `startTimer(T3)` will be `prepend`ed in front of `startTimer(T2)`, `startTimer(T3)` will not get executed for this invocation of the I/O function. However, had we instead been `append`ing, the `startTimer(T3)` will get executed for this invocation.



4 ENTRY-RETURN INSTRUMENTATION

In the previous sections, we looked at the basic properties of counters and timers. In this section, we look at how we can properly update the counters and timers in our entry-point and return-point instrumentation. Section 5 will cover callsite instrumentation.

4.1 Return-point execution does <u>NOT</u> imply entry-point execution

The simplest example is the case where the program was executing within the function when the function becomes instrumented. The entry-point instrumentation will not get executed for this invocation of the function because the entry point has already been passed. However, the return-point instrumentation may get executed as part of the function's return. We use an example to illustrate how to handle this situation.

Here are the most obvious MDL statements for an inclusive constraint flag. `constraintFlag` is nonzero if there is any known active invocation of the instrumented function.

```
counter constraintFlag;

append preInsn func.entry  (* constraintFlag += 1; *)
append preInsn func.return (* constraintFlag -= 1; *)
```

Let us make a simple walk-through of the above MDL statements. As with all counters, `constraintFlag` is initialized to zero. On entry to the instrumented function, `constraintFlag` is incremented to 1. On return from the instrumented function, `constraintFlag` is decremented back to 0. Should the instrumented function be recursive, the recursive invocation will first increase `constraintFlag` on entry, but will properly restore `constraintFlag` on return. It seems that the above MDL statements correctly give `constraintFlag` a nonzero value when there is an active invocation of the function.

However, suppose that the program was executing within the function at the moment of instrumentation. Then the entry-point instrumentation will not be executed, but the return-point instrumentation is. The result is that `constraintFlag` gets set to -1 at return. Now `constraintFlag` is nonzero even when there is no active invocation of the function. It gets worse. On the next invocation of the function, the entry-point instrumentation will increase `constraintFlag` to zero, and the return-point instrumentation will decrease `constraintFlag` back to -1. Therefore, `constraintFlag` is zero when there is a known active invocation of the function. If this function is nonrecursive, our instrumentation has failed in every case.

To correct the situation, here is a better MDL implementation. `constraintFlag` is nonzero if and only if it has been incremented at entry to the function. Using this knowledge, the new implementation executes the return-point instrumentation only if the entry-point instrumentation has been executed. Another way to look at it is that we simply do not decrement `constraintFlag` below zero. Of course, if instrumentation occurred while the program was executing within the function, `constraintFlag` may still have the incorrect value of zero for the duration of the function's current invocation.

```
// Being careful in return-point instrumentation

counter constraintFlag;

append preInsn func.entry  (* constraintFlag += 1; *)
append preInsn func.return (* if (constraintFlag != 0) constraintFlag -= 1;*)
```

4.2 Entry-point execution does NOT imply return-point execution

Some functions will return abnormally. For example, suppose there was an exception. Some languages with exception handling will allow a function to unwind the callstack to an arbitrary depth until it finds a function that can handle the exception. Another example is the `longjmp()` library function from the C programming language. Calling this function essentially unwinds the call-stack. Yet another example is a case where we did not detect an instruction sequence as a function-return sequence. There are many ways to return from a function. With optimizing compilers and creative assembly programmers, we do not claim to know all the instruction sequences that may be used to return from a function. As a result, we may not have instrumented all return points of a function.

Let us look at a timer example. Here are the most obvious MDL statements to measure the execution time of a function. Timer `T` is on if and only if there is a known active invocation of the function.

```
append preInsn func.entry  (* startProcessTimer(T); *)
append preInsn func.return (* stopProcessTimer(T);  *)
```

Let us take a simple walk-through of the above code. On entry to the function, timer `T` is started. On return from the function, timer `T` is stopped. If the program was executing within the function at the moment of instrumentation, then we will simply stop a timer that was not started, which has no effect. Things look good so far.

However, suppose there was a normal entry to the function (timer `T` started) followed by a rarely used exceptional return (timer `T` not stopped). Remember that timers are stopped only if all `startTimer()` calls are matched by `stopTimer()` calls. In this case, the `startProcessTimer(T)` at entry was not matched by the `stopProcessTimer(T)` at return, so timer `T` is still running. On the next invocation of the function, we execute a new `startProcessTimer(T)`, so we now have two unmatched `startProcessTimer(T)`. At normal return, we execute `stopProcessTimer(T)` to match the new `startProcessTimer(T)`, but the other unmatched `startProcessTimer(T)` is still unmatched. So at return, the timer `T` is still running. In fact, timer `T` will always remain running, regardless of whether or not the function is actually being executed.

To solve this problem, we write a better MDL implementation that has a "self-healing" property. The next normal execution of the function will correct the instrumentation problems of the previous exceptional execution of the function[1]. We accomplish this by making sure that there is never more than one unmatched `startProcessTimer(T)`. The new MDL code follows. (Unfortunately, there is no simple way to extend this technique to recursive functions.)

```
// A self-healing timer metric

counter T_is_running;  // Flag is 1 if timer T is running

append preInsn func.entry
  (* if (T_is_running == 0) startProcessTimer(T);
     T_is_running = 1;
   *)

append preInsn func.return
  (* T_is_running = 0;
     stopProcessTimer(T);
```

1. Note, however, that this "healing" of the instrumentation problem such that subsequent execution will be correct, doesn't correct or compensate for spurious accounting which takes place in the interrim.

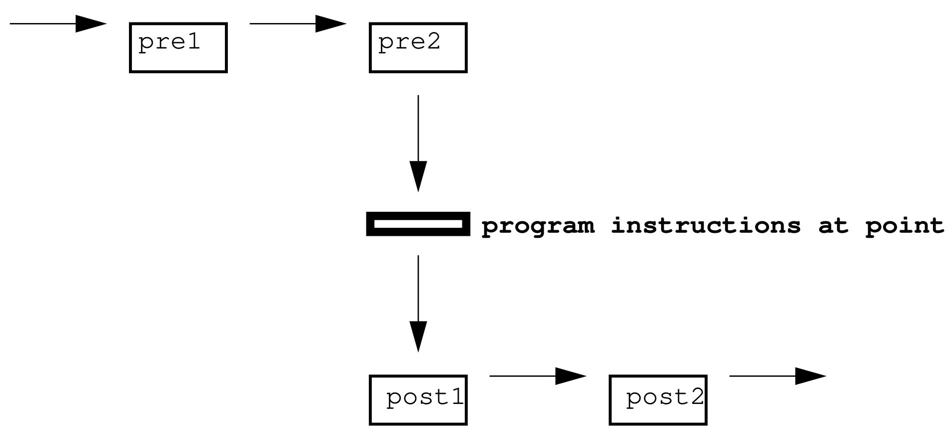
```
    *)
```

We are using the counter `T_is_running` as a flag that is true when there is an unmatched `startProcessTimer(T)`. `T_is_running` is correctly initialized to zero since the timer `T` is not running at the moment of instrumentation. At entry to the function, we turn the timer on if and only if it has not already been turned on. Then we set `T_is_running` to 1 to indicate that at this point we are sure the timer is running. Notice that no matter how many times you repeatedly execute this entry-point instrumentation, there is at most one `startProcessTimer(T)` that actually gets executed. At normal return from the function, the `stopProcessTimer(T)` will match the single unmatched `startProcessTimer(T)`. We also set `T_is_running` to zero since we are sure there will be no unmatched `startProcessTimer(T)`.

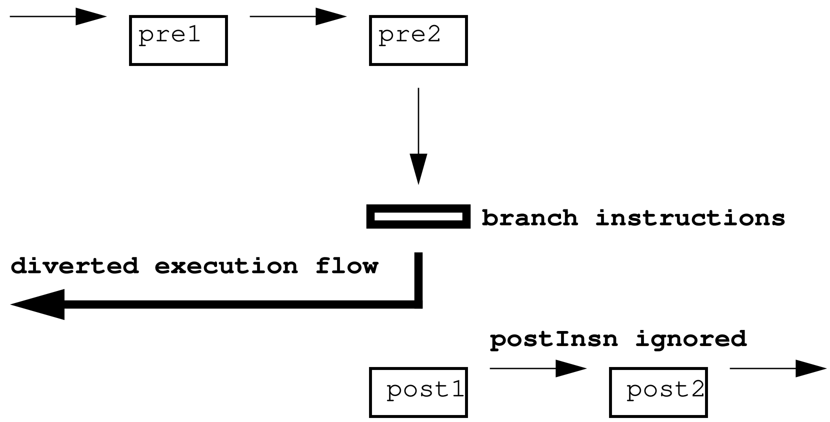
Let us look at the case where we enter the function normally, execute `startProcess-Timer(T)`, and return exceptionally without executing the return-point instrumentation. Now the timer `T` is incorrectly running, and the `T_is_running` flag is still set to 1. On the next invocation of the function, the `startProcessTimer(T)` is not executed. If the function returns normally this time, the `stopProcessTimer(T)` will correctly turn off the timer, and the `T_is_running` flag will be correctly set to 0. Our instrumentation is now "healed".

4.3 Avoid using entry-point post-instrumentation

The sequence in which Paradyn executes instructions at an instrumentation point is as follows. First, any pre-instrumentation is executed, then the program instructions located at the instrumentation point, and afterwards, any post-instrumentation is executed. This is illustrated below.



You should try to avoid using entry-point post-instrumentation because the program instructions found at the point may be branch instructions. These instructions will divert execution flow away from the post-instrumentation, as illustrated in the figure below. When this happens, any post-instrumentation will not be executed. In practice, we have not found it necessary to use entry-point post-instrumentation.



Therefore, you should avoid using any MDL statements that look like the following. Future versions of Paradyn may even treat them as syntax errors.

```
append  postInsn func.entry  ...   // RISKY!
prepend postInsn func.entry  ...   // Just as RISKY!
```

4.4 Do NOT use return-point post-instrumentation

Return-point post-instrumentation does not work and does not make sense. Therefore, Paradyn treats any attempt to use such instrumentation as a syntax error.

```
append  postInsn func.return  ...   // Syntax ERROR!
prepend postInsn func.return  ...   // Syntax ERROR!
```

5 CALLSITE INSTRUMENTATION

Callsite instrumentation is commonly used in exclusive timer metrics. These metrics measure the execution time of a function, but they must exclude the time spent in functions *called* by the function being timed. Here is a typical example of MDL code for an exclusive timer.

```
counter T_is_running;  // Flag is 1 if timer T is running

append preInsn func.entry
  (* if (T_is_running == 0) startProcessTimer(T);
     T_is_running = 1;
   *)

append preInsn func.return
  (* T_is_running = 0;
     stopProcessTimer(T);
   *)

foreach callsite in func.calls {
  append preInsn  callsite
    (* T_is_running = 0;
       stopProcessTimer(T);
     *)

  append postInsn callsite
    (* if (T_is_running == 0) startProcessTimer(T);
       T_is_running = 1;
     *)
}
```

Let us take a simple walk-through. At entry to the function, the timer is started. At return from the function, the timer is stopped. This times the execution of the function. Now we proceed to exclude the execution times of functions called by this function. We accomplish this by instrumenting the callsites. At each callsite, we stop the timer before entering the called function, and we restart the timer after returning from the called function.

Notice in the MDL code given above that the callsite pre-instrumentation is the same as the function return-point instrumentation and that the callsite post-instrumentation is the same as the function entry-point instrumentation. In general, this is true for all exclusive metrics.

5.1 In exclusive metrics, `preInsn callsite` is analogous to `func.return`

At a callsite, the function being measured is going to temporarily stop executing until the called function returns. At function return, the function is going to temporarily stop executing until the next invocation of the function. To an exclusive metric, the difference in why the function is stopping execution is irrelevant. All that matters is that the exclusive metric must halt until the function resumes execution.

5.2 In exclusive metrics, `postInsn callsite` is analogous to `func.entry`

Immediately following a callsite, the function being measured is going to resume execution after temporarily stopping to let another function execute. At function entry, the function is going to resume execution after being temporarily stopped between invocations. To an exclusive metric, the difference in why the function is resuming execution is irrelevant. All that matters is that the exclusive metric must be restarted until the function stops execution again.

6 LOOP INSTRUMENTATION

MDL supports loops as resources within a function. Instrumentation can be inserted into any one of four places in a loop: iteration start, iteration end, loop entry, and loop exit. Iteration start instrumentation executes once every time the loop starts a new iteration, and iteration end instrumentation executes once every time a loop finishes an iteration. These two points are guarenteed to match up, so each start iteration is guarenteed to be followed by an end iteration (presuming the loop isn't infinite or terminates the program). Loop entry instrumentation executes once every time the loop is entered, and loop exit instrumentation executes once when the loop finishes (whether that is from the looping condition being met, or a premature exit as if from a break statement).

6.1 Loop Names

Loops are named based on their nesting level and the function that contains them. If a function has n loops, then they're named loop_1 through loop_n. Inner loops are named based on the loop that contains them; if loop_1 contains three inner loops, then those three are named loop_1.1, loop_1.2, and loop_1.3.

Consider this example code from `foo.c`

```
void func_1() {
  for (int A=0; A<10; A++)
    if (A==5) break;
}

void func_2() {
  for (int B=0; B<15; B++)
    for (int C=0; C<5; C++)
      printf("%d, %d\n", B, C);
  for (int D=0; D<10; D++)
    printf("%d\n", D);
}
```

The A loop in func_1() is named by /foo.c/func_1/loop_1. The B and D loops in func_2 are named by /foo.c/func_2/loop_1 and /foo.c/func_2/loop_2. The B loop also contains an inner loop, loop C, which is referred to as /foo.c/func_2/loop_1.1. Loops can also be referred to by wildcards. So /foo.c/*/* means all loops in foo.c. /foo.c/func_2/* refers to all loops in func_2.

6.2 Inserting Instrumentation into Loops

Instrumentation can be inserted into points at iteration begin, iteration end, loop entry, and loop exit. These are respectively referred to as start_iter, end_iter, enter, and exit points in MDL. The following metric gives and example of how to count the number of iteration a loop performs:

```
metric loopIterations {
  name "loop_iters";
  units ops;
  aggregateOperator sum;
  style EventCounter;
  flavor { winnt, unix, mpi };
  unitsType unnormalized;

  constraint loopConstraint /Code/*/* is replace counter {
    prepend preInsn $constraint[0].start_iter (* loopIterations++; *)
    append  preInsn $constraint[0].end_iter (* loopIterations++; *)
  }

  base is counter {
  }
}
```

This metric gives an example of how to count the number of times a loop is executed:

```
metric loopExecutions {
  name "loop_executions";
  units ops;
  aggregateOperator sum;
  style EventCounter;
  flavor { winnt, unix, mpi };
  unitsType unnormalized;

  constraint loopConstraint /Code/*/* is replace counter {
    prepend preInsn $constraint[0].enter (* loopExecutions++; *)
    append  preInsn $constraint[0].exit  (* loopExecutions++; *)
  }

  base is counter {
  }
}
```

7 MISCELLANEOUS

This section contains MDL programming tips that could not nicely fit within any of the previous sections and that could not warrant a separate section by themselves.

7.1 All EventCounter-style metric values must increase monotonically

The values of your EventCounter metrics should not be allowed to decrease in value, no matter how slightly or for how short an amount of time.[2] Paradyn rigorously checks this condition and will immediately abort if the check ever fails. A typical example of such a metric is one that accumulates a count of the number of bytes transferred by the I/O functions. Sample MDL code is given below. The I/O functions give the number of bytes transferred as the return value.

```
// Blindly adding function return values

counter in_sampling;      // Flag is 0 if DYNINSTalarmExpire() is not
                          //  currently being executed

foreach func in io_funcs {
  prepend preInsn func.return constrained
```

2. SampledFunction-style metrics, of which an example is provided in Section 7.2, allow reported values to change arbitrarily, however, they must still remain non-negative.

```
    (* if (in_sampling == 0) io_bytes += $return; *)
}


foreach func in DYNINSTalarmExpire {
  prepend preInsn func.entry
    (* in_sampling = 1; *)

  append preInsn func.return
    (* in_sampling = 0; *)
}
```

However, the I/O functions also return -1 if they encounter errors. In fact, it is a general practice in C programming for functions to return negative values to indicate errors. Therefore, whenever an I/O function returns due to error, our instrumentation will add a *negative* number to the metric value counter `io_bytes`. Paradyn will detect this decrease in the metric's value, and immediately abort, ending your application's performance measurement session with it.

To solve this problem we simply check the numbers before adding them to their respective metric value counters. The MDL code from above is corrected below by adding such a check.

```
// Careful when adding function return values

counter in_sampling;      // Flag is 0 if DYNINSTalarmExpire() is not
                          //  currently being executed

foreach func in io_funcs {
  prepend preInsn func.return constrained
    (* if ((in_sampling == 0) && ($return > 0)) io_bytes += $return; *)
}

foreach func in DYNINSTalarmExpire {
  prepend preInsn func.entry
    (* in_sampling = 1; *)

  append preInsn func.return
    (* in_sampling = 0; *)
}
```

7.2 Use SampledFunction-style metrics to track values in a user application

One use of MDL likely to be of particular interest to application or library developers, is the ability to query a program value and track its evolution. This is generally achieved in two steps: specification of a query function in the user program which returns the value of interest (similar to those common for returning the values of private members of C++ classes), and a specification of when this value should be queried when the metric is enabled.

Because Paradyn uses a shared-memory sampling approach for efficiently extracting performance data from the application, a suitable user function needs to be identified as the point when values should be sampled: this may well be the function (or list of functions) which actually update the value of interest.

The following example functions from a user program query and update a program value (which in this case is a global variable, but could have been accessed by any other means):

```
unsigned int value;
unsigned int program_value() { return (value); }
void update_program_value (unsigned int new_value) { value=new_value; }
```

The metric itself should be specified of style `SampledFunction` (and unitsType `sampled`), such that the current value is available, rather than the delta from the previously sampled value. This also has the advantage of allowing values to both increase and decrease, rather than restricting the value to increase monotonically as is the case with the EventCounter style of metric. Note, however, that sampled values need to be unsigned integers.

```
// sample/report a program value after it is updated in a user program

resourceList update_function_list is procedure {
        items { "update_program_value" };        // user's update function
        flavor { unix };
        library false;
}

metric programValue {
       name "program_value";
       style SampledFunction;
       units value_units;
       unitsType sampled;
       aggregateOperator sum;
       flavor { unix };

  base is counter {
    foreach func in update_function_list {
      append preInsn func.return constrained
        (* programValue = program_value(); *)      // user's sampling function
    }
  }
}
```

Instead of invoking the user function `progam_value()` to obtain the value of interest, MDL provides the specially-defined `readSymbol` query function to directly read a global variable:

```
(* programValue = readSymbol("value"); *)     // read global symbol value
```

Finally, it is worth noting that while the requested value is updated and stored as specified in the application program's space, sampling of this value and its reporting by the Paradyn daemon to the Paradyn front-end happens completely asynchronously. A consequence of this will be that Paradyn will generally miss data value updates (or repeatedly re-sample the same value) leading to spurious accumulations (in totals or averages) when compared to sample accumulations by the user program itself. At best, the reported samples are an approximation of the actual sample updates requested.

7.3 Instrumenting recursive functions

Recursive functions present particular difficulties for instrumentation metrics, and a number of the techniques presented in this guide do not readily apply to this class of funtions, or become considerably more complicated when they must robustly deal with cases of already executing recursive functions and exceptional returns.

Particular care (and experimentation) is required writing metrics for these cases.

8 CONCLUSION

This paper was written to help you become more proficient in programming with Paradyn's Metric Description Language (MDL). Specifically, it was designed to save you time and frustrations in using this new and unusual language. We discussed the two basic MDL metric value types: counters and timers. We covered tips on how to properly update them in instrumentation. We also covered how to keep metric values from decreasing.

We have tried to keep this document brief and readable so that you may be encouraged to read it in its entirety. For more MDL examples, the best source is the `paradyn.rc` configuration file provided to you. However, some metrics in the `paradyn.rc` were implemented using in-depth knowledge about the functions and/or computing platforms involved, so some of the programming tips developed in this paper were skipped without affecting the correctness of those metrics. Please keep in mind that such discrepancies do not invalidate the advice given in this paper since you may not have (or want to acquire) such detailed information about your computing platform.

To simplify the presentation, we provided only MDL code fragments, and not complete metric implementations. In addition, though *constraints* are key to instrumentation, none of the MDL examples given were `constrained`. All examples were written as if for whole-program metrics. Again, this was done to simplify the text. The metrics you write, however, will usually need to be `constrained`.

Thank you for your interest in Paradyn. We hope this paper was useful to you and that each future reading continues to provide you new insights. If you have additional questions, please send email to `paradyn@cs.wisc.edu`.

n