Paradyn Parallel Performance Tools

Instrumentation of Multithreaded Programs

Release 3.3
January 2002

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

Table Of Contents

1 INTRODUCTION

The purpose of this document is to describe all the general issues related to the implementation of the instrumentation of threaded programs using Paradyn. Here, we will explain what the model is, what data structures are used, how the instrumentation actually works and what the current problems and limitations are.

Although most of the ideas described here are general, this design is based on the Solaris Multithreaded Architecture (Figure 1). In this model, threads are user level structures that are assign to light weight processes (LWPs). The LWPs are kernel threads and these are assign to actual CPUs. The LWPs are also known as virtual CPUs. For more information, please look at the documentation about Solaris threads.

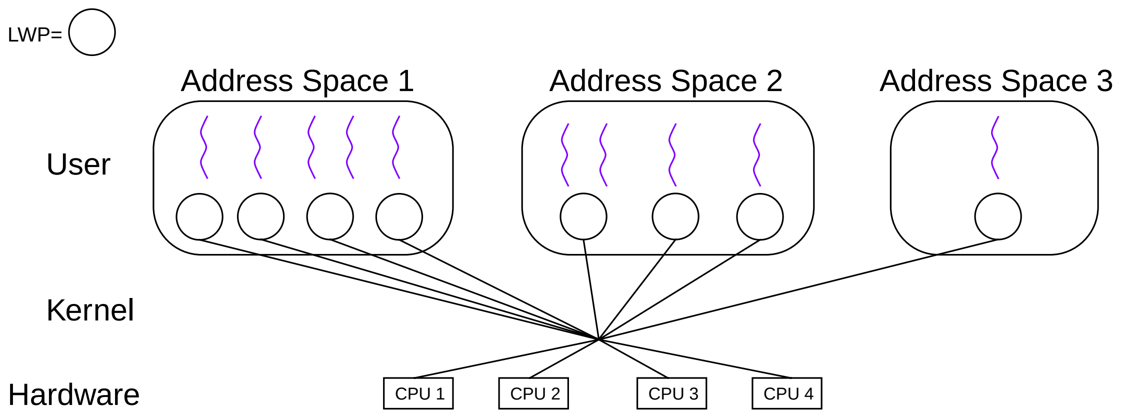

Figure 1: Solaris Multithreaded Architecture

2 PARADYN PROGRAM INSTRUMENTATION

In the single threaded version of Paradyn, we used to have instrumentation code and data (counters or timers) on a per process basis. Whenever we created a new metric, we would insert instrumentation for a particular process and create the corresponding counter/timer.

With multithreaded applications, the situation is quite different. Every time we modify the program’s image, we know that all threads can execute this instrumentation so we need to make sure that this instrumentation computes the right thing on a per thread basis. In order to do this, we use a modified version of the instrumentation code that is very similar to the single threaded case but that uses the current thread id of the executing thread to know what data to access. In this way and having counter/timers per thread, we can compute metrics for any single thread without having specific instrumentation for any special thread (see Figure 2)

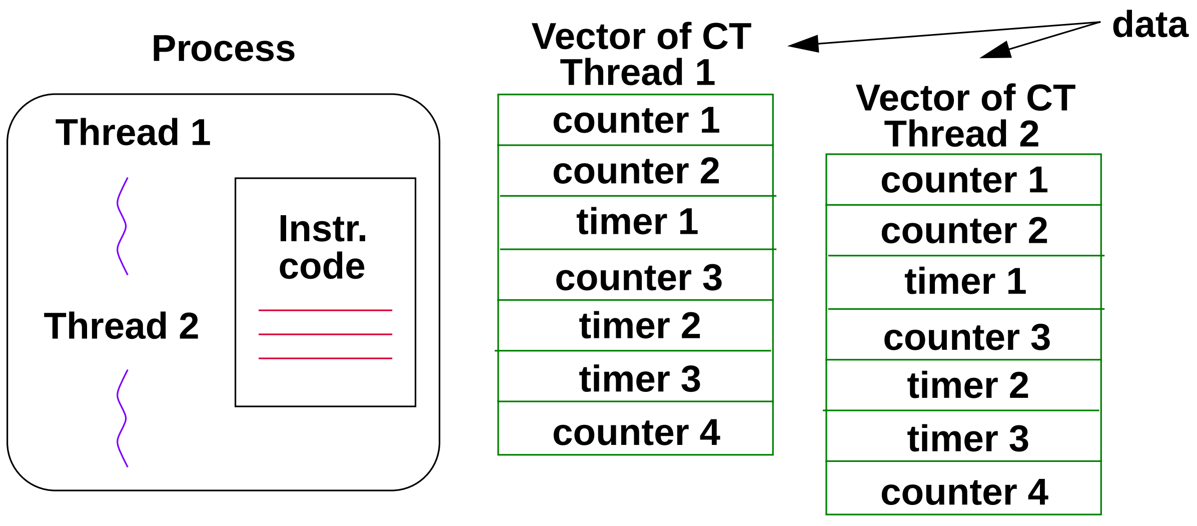

Figure 2: Instrumenting multithreaded applications. Single instrumentation code and multiple data per thread.

3 DESIGN ISSUES

Some of the most important design issues are:

- ❑ Every thread shares the same program instrumentation.
- ❑ There is a vector of counter/timers per thread (more memory usage but better speed and more straight forward implementation).
- ❑ Two base application scenarios: few threads, few LWP (light weight processes); many threads created and destroyed dynamically (e.g. servers).

4 CURRENT DESIGN

The current design to support multithreaded applications with Paradyn can be seen in Figure 3.

The main structure is the *Thread Table*, which can be viewed as a matrix. The columns of this metric represent threads and the rows metrics. Each cell in the matrix is a pointer to a *Vector of Counter* or *Timers*. These vectors contain a set of counter/timers (a fixed number actually) and since they are stored inside these vectors we don't need any other level of indirection to access the data. To clarify how we access data using this data structure, let's give an example. If we want to access counter C4 for thread N, first we need to know the thread id of thread N. In this way we will be able to access the column corresponding to thread N in the Thread Table. With this information, we can then access the right vector of counters for thread N (base address of the vector). Now, we need to know the offset or position on this vector (in this example, C4). The offset or position in the vector is the same for every thread. This condition is very important because it

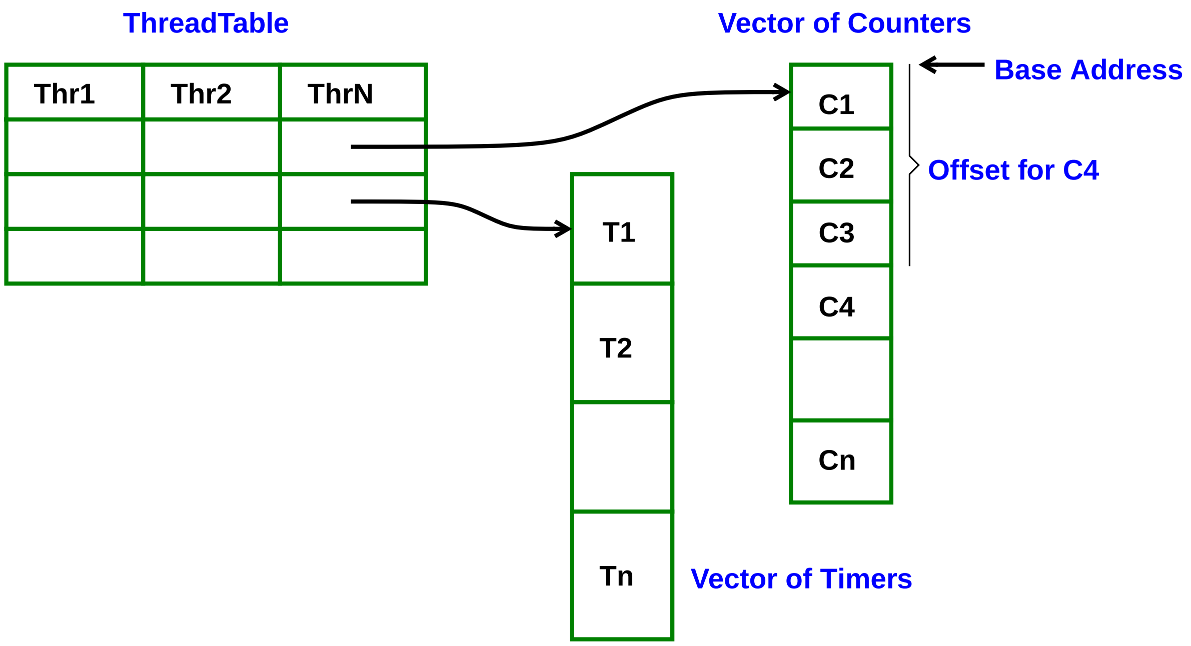

Figure 3: Basic data structure for instrumenting threaded programs.

allow us to keep the same instrumentation code for every thread (i.e. all threads *share* this code). In summary, to access a particular counter/timer we need three values: the thread id, the row number in the thread table (usually called level, which actually tell us what metric we are accessing and whether it is a counter or timer) and finally the offset or position in the vector.

This structure is represented in `paradynd` by the following classes:

- `superTable`. The `superTable` class consists of an array of `baseTable` elements (`superVectors`) and it represents the *Thread Table* in paradynd. The `superTable` class is the class that has contact with the outside world. Rows on this table are represented by `superVectors`. Each `superVector` has one entry for each thread (although we could have more entries than threads, in which case we call these entries "reserved"). In order to know what entry correspond to what thread, we use a hash table (which has a counter part in the application). The `superTable` has also different "levels". Each level is used to store a different kind of `dataReqNode` (i.e. counters, wall timers and process timers). When we add a new counter, we need to find first on which level or levels are we storing counters. Then we need to find an empty position in the `fastInferiorHeap`. Each entry of the `superVector` has a `fastInferiorHeap` associated with it. This structure has the mapping of data elements to locations in the shared memory segment. This mapping is the same for every entry in the `superVector` (i.e. for every thread), which means that when we add a new data element, we are adding it to every thread. The reason for this is that having the same offset for counters of different threads will make the instrumentation code a lot easier and faster.
- `baseTable`. The `baseTable` class consists of an array of `superVectors`. The `baseTable` class is a template class. It has a `levelMap` vector that keeps track of the levels (rows) that has

been allocated (remember that we need not only an index but also a level in order to determine the position of a counter or timer).

- `superVector`. The `superVector` is an array of vectors of counters and timers, or `fastInferiorHeap` objects. Part of the functionality of the `fastInferiorHeap` class has been moved to this new class.

Figure Figure 4, provides a general view of all these classes and how they are related.

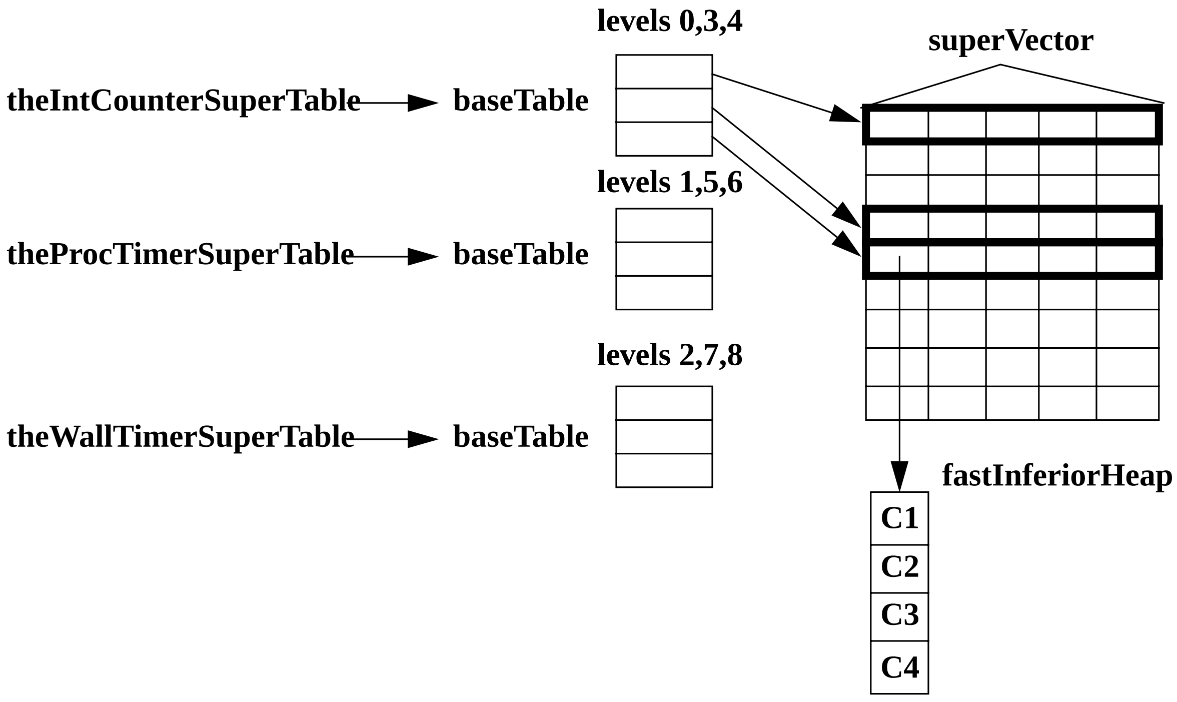

Figure 4: superTable, baseTable, superVector and fastInferiorHeap classes.

4.1 Whole process vs. Threads

So far we have figured out how are we going to measure metrics for a particular thread, but how are we going to handle metrics for the a particular process? We have several ways of doing this and some could be more efficient than others. One option is to compute these metrics in the old good way: having a single counter/timer for the whole process. In this way, we wouldn't need to aggregate values for all threads but we would have to put locks every time we update the counter/timer since all threads are going to update the same variable. The other alternative, which is the one we choose because we think is the cheapest and more "natural" one, is to always compute counter/timers per thread and then aggregate all the values to get a per process metric. Using this approach, we don't need locks but we do need to aggregate values all the time. If the number of threads is large, this operation could be expensive plus we need to keep adding and deleting threads to this metric every time a thread is actually created or deleted.

4.2 Instrumentation Details

Some of the most important instrumentation issues are:

- ❏ Fixed size Thread Table indexed by thread-ids, points to vectors of counters or timers.
- ❏ Thread Table size directly related to maximum number of active threads at any given time.
- ❏ Separate sets of vectors of counters and timers per thread.
- ❏ Creation of vectors of counters/timers on demand, never removed.
- ❏ Counter/timers allocated by blocks (2 level memory allocation).

4.3 Base Trampoline

Figure 5 shows an updated version of the base trampoline to instrument multithreaded applications. There are two new sections in the base trampoline. The first one is called “MT Preamble” and its main function is to compute the position in the thread table for the thread that is currently executing the code. Once we compute this value, we will store it in a special register (l7 more specifically). In this way, we will guarantee that this value is going to be different for different threads since registers are kept per thread (and we don’t need any special thread local storage). The second new section is located in the mini-trampolines and it computes the address of the counter/timer based on the offset (we will discuss this in the following section).

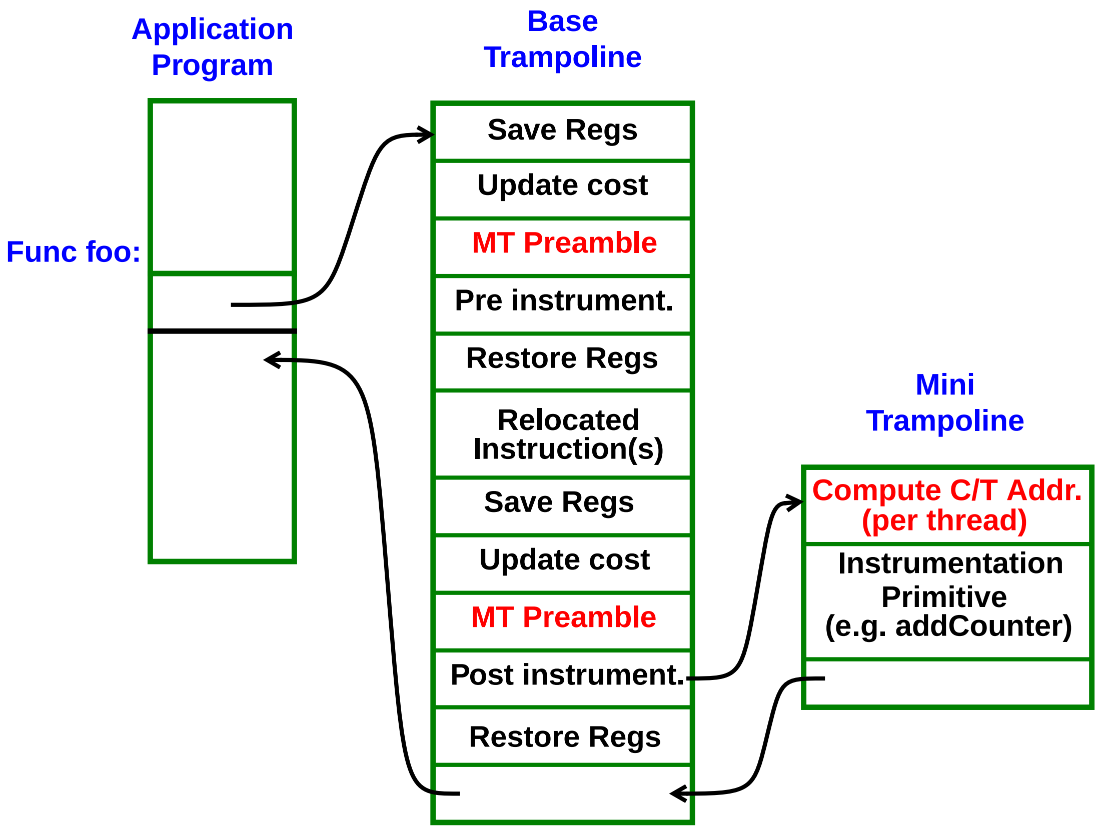

Figure 5: Modified base trampoline.

An example of the assembly code for a MT Preamble in a base trampoline (Sparc Solaris Architecture) can be found in Figure 6. The instructions that comprise MT Preamble are in bold

```
0x23090:        save  %sp, -120, %sp
0x23094:        nop
0x23098:        nop
0x2309c:        nop
0x230a0:        std  %g0, [ %fp + -8 ]
0x230a4:        std  %g2, [ %fp + -16 ]
0x230a8:        std  %g4, [ %fp + -24 ]
0x230ac:        std  %g6, [ %fp + -32 ]
0x230b0:        sethi  %hi(0xee40f800), %o5
0x230b4:        call  %o5 + 0x3e0       ! 0xee40fbe0 <DYNINSTthreadPos>
0x230b8:        nop
0x230bc:        cmp  %o0, -2
0x230c0:        mov  1, %l0
0x230c4:        bne,a   0x230cc
0x230c8:        clr  %l0
0x230cc:        cmp  %l0, 0
0x230d0:        be  0x230e0
0x230d4:        nop
0x230d8:        b,a   0x23124
0x230dc:        nop
0x230e0:        sll  %o0, 2, %l0
0x230e4:        sethi  %hi(0xeeb25800), %l1
0x230e8:       or  %l1, 0x13c, %l1     ! 0xeeb2593c <DYNINSTthreadTable>
0x230ec:        add  %l0, %l1, %l0
0x230f0:        mov  %l0, %l7
0x230f4:        nop
0x230f8:        nop
0x230fc:        nop
0x23100:        nop
0x23104:        nop
0x23108:        nop
0x2310c:        nop
0x23110:        nop
0x23114:        nop
0x23118:        nop
0x2311c:        b,a   0x23230
0x23120:        nop
0x23124:        ldd  [ %fp + -8 ], %g0
0x23128:        ldd  [ %fp + -16 ], %g2
0x2312c:        ldd  [ %fp + -24 ], %g4
0x23130:        ldd  [ %fp + -32 ], %g6
0x23134:        sethi  %hi(0xedc00000), %l0
0x23138:        ld  [ %l0 + 0xc ], %l1  ! 0xedc0000c
0x2313c:        add  %l1, 0x54, %l1
0x23140:        nop
0x23144:        nop
0x23148:        st  %l1, [ %l0 + 0xc ]
0x2314c:        restore
```

Figure 6: Example of Base Trampoline. MT Preamble, pre-instrumentation.

face. The call to `DYNINSTthreadPos` returns the column index of the calling thread in the ThreadTable, and `DYNINSTthreadPos` returns -2 if the execution of the thread has already passed the point where we have put instrumentation code to detect thread termination (i.e., thread has terminated from the tool’s perspective).

4.4 Mini-Trampolines

Mini-trampoline code is similar to the single threaded version, except that it needs to compute the address of the counter/timer before executing the desired operation (e.g. increment a counter). This computation of the address also involves the computation of the address for the vector of counter/timers.

Figure 7 shows an example of mini-trampoline code that increments a counter. In Figure 7, counter is at level 0, with an offset 0 in the corresponding the vector of counters (implemented as a fastInferiorHeap). The instruction at line 0x23230 computes the base address of the fastInferiorHeap, and the instructions at line 0x23234-0x23248 checks whether the base of the fastInferiorHeap is NULL. The instruction at line 0x23250 computes the address of the counter by adding the offset to the base of the fastInferiorHeap.

```
0x23230:        ld  [ %l7 ], %l0
0x23234:        cmp  %l0, 0
0x23238:        mov  1, %l1
0x2323c:        be,a   0x23244
0x23240:        clr  %l1
0x23244:        cmp  %l1, 0
0x23248:        be  0x23260
0x2324c:        nop
0x23250:        add  %l0, 0, %l0
0x23254:        ld  [ %l0 ], %l1
0x23258:        inc  %l1
0x2325c:        st  %l1, [ %l0 ]
0x23260:        b,a   0x23270
```

Figure 7: Mini-trampoline code. Counter. Loads vector address (in this case level is 0, but there could be some computation first), check that the value is not NULL, uses the offset to compute counter address, loads counter, increments counter, saves counter and returns.

Figure 8 shows an example of mini-trampoline that starts a Thread timer. The mini-trampoline first checks if the thread has already terminated (from the perspective of Paradyn), and it then loads the base address of the vector of timers (see the instructions at lines 0x23288-0x23294, and in this case the timer is stored in a vector at level 2).

Paradyn allocates the vector of timers (implemented as fastInferiorHeaps) for a new thread asynchronously when it is notified the thread has been created. It is possible that before Paradyn have the chance to allocate the fastInferiorHeaps for the new thread, the application could have already entered an instrumentation code. To deal with this, the mini-trampoline spins until the base address of the corresponding fastInferiorHeap becomes non-NULL.

4.5 InferiorRPC

Paradyn uses an inferiorRPC to execute code that would have not been executed otherwise (e.g. starting a timer for the whole program *after* the program has already started). The problem of doing this operation on a multithreaded program, is that we would need to execute this code for a particular thread (i.e. the thread that requires to start a timer, for example). One alternative to this is that the main thread (or any other thread) executes the code *on behalf* of the particular thread that needs to run the inferiorRPC. We achieve this by forcing the use of a particular thread id in

```
0x23270:        sethi  %hi(0xee40fc00), %o5
0x23274:        call  %o5 + 0x9c        ! 0xee40fc9c <DYNINST_not_deleted>
0x23278:        nop
0x2327c:        cmp  %o0, 0
0x23280:        be  0x232f0
0x23284:        nop
0x23288:        sethi  %hi(0x1000), %l2
0x2328c:        mov  %l2, %l2   ! 0x1000
0x23290:        add  %l7, %l2, %l1
0x23294:        ld  [ %l1 ], %l0
0x23298:        cmp  %l0, 0
0x2329c:        mov  1, %l1
0x232a0:        bne,a   0x232a8
0x232a4:        clr  %l1
0x232a8:        cmp  %l1, 0
0x232ac:        be  0x232dc
0x232b0:        nop
0x232b4:        sethi  %hi(0xee40fc00), %o5
0x232b8:        call  %o5 + 0x8c        ! 0xee40fc8c <DYNINSTloop>
0x232bc:        nop
0x232c0:        cmp  %o0, 0
0x232c4:        be  0x232d4
0x232c8:        nop
0x232cc:        b,a   0x232f0
0x232d0:        nop
0x232d4:        b,a   0x23288
0x232d8:        nop
0x232dc:        add  %l0, 0, %l0
0x232e0:        mov  %l0, %o0
0x232e4:        sethi  %hi(0xee40a400), %o5
0x232e8:       call  %o5 + 0x368      ! 0xee40a768 <DYNINSTstartThreadTimer>
0x232ec:        nop
0x232f0:        b,a   0x23120
```

Figure 8: Mini-trampoline code. Timer. Loads vector address (in this case, the level is different than 0, so we need to add some offset to the base address of the Thread Table), check that the value is not NULL, uses the offset to compute timer address, calls `DYNINSTstopProcessTimer(timer_address)` and returns.

the base-trampoline (the place where we compute what thread is currently executing the code). The following paragraph describes some of the details of the assembly code for an inferiorRPC.

This is an example of the assembly code for the inferiorRPC (Figure 9). Remember that in this case, we need to execute this code on behalf of another thread.

A brief explanation of Figure 9 goes as follows:

The first instruction creates a new frame for the inferiorRPC. The call to `DYNINSTthread-PosTID` takes the thread id, and the position of the thread in the ThreadTable and tests to see if the thread has already terminated (from the perspective of Paradyn). It then executes essentially the same instructions as that would have been in a mini-trampoline, except that all calls takes an extra parameter, and it tests if a thread has terminated before every major operations, since the thread that carries out the inferiorRPC may be different from the threads which the inferiorRPC is intended for. The complications of this code mainly result from the goal for generality.

```
0xefd9b180:     save  %sp, -112, %sp
0xefd9b184:     sethi  %hi(0), %l0
0xefd9b188:     or  %l0, 5, %l0 ! 0x5
0xefd9b18c:     sethi  %hi(0), %l1
0xefd9b190:     or  %l1, 5, %l1 ! 0x5
0xefd9b194:     mov  %l0, %o0
0xefd9b198:     mov  %l1, %o1
0xefd9b19c:     sethi  %hi(0xee408c00), %o5
0xefd9b1a0:     call  %o5 + 0x310       ! 0xee408f10 <DYNINSTthreadPosTID>
0xefd9b1a4:     nop
0xefd9b1a8:     cmp  %o0, -2
0xefd9b1ac:     mov  1, %l0
0xefd9b1b0:     bne,a   0xefd9b1b8
0xefd9b1b4:     clr  %l0
0xefd9b1b8:     cmp  %l0, 0
0xefd9b1bc:     be  0xefd9b1cc
0xefd9b1c0:     nop
0xefd9b1c4:     b,a   0xefd9b288
0xefd9b1c8:     nop
0xefd9b1cc:     sll  %o0, 2, %l0
0xefd9b1d0:     sethi  %hi(0xeeb25800), %l1
0xefd9b1d4:     or  %l1, 0x13c, %l1     ! 0xeeb2593c <DYNINSTthreadTable>
0xefd9b1d8:     add  %l0, %l1, %l0
0xefd9b1dc:     mov  %l0, %l7
0xefd9b1e0:     sethi  %hi(0), %l0
0xefd9b1e4:     or  %l0, 5, %l0 ! 0x5
0xefd9b1e8:     sethi  %hi(0), %l1
0xefd9b1ec:     or  %l1, 5, %l1 ! 0x5
0xefd9b1f0:     mov  %l0, %o0
0xefd9b1f4:     mov  %l1, %o1
0xefd9b1f8:     sethi  %hi(0xee408800), %o5
0xefd9b1fc:     call  %o5 + 0x3b8       ! 0xee408bb8 <DYNINST_not_deletedTID>
0xefd9b200:     nop
0xefd9b204:     cmp  %o0, 0
0xefd9b208:     be  0xefd9b288
0xefd9b20c:     nop
0xefd9b210:     sethi  %hi(0x1000), %l4
0xefd9b214:     mov  %l4, %l4   ! 0x1000
0xefd9b218:     add  %l7, %l4, %l3
0xefd9b21c:     ld  [ %l3 ], %l2
0xefd9b220:     cmp  %l2, 0
0xefd9b224:     mov  1, %l3
0xefd9b228:     bne,a   0xefd9b230
0xefd9b22c:     clr  %l3
0xefd9b230:     cmp  %l3, 0
0xefd9b234:     be  0xefd9b26c
0xefd9b238:     nop
0xefd9b23c:     mov  %l0, %o0
0xefd9b240:     mov  %l1, %o1
0xefd9b244:     sethi  %hi(0xee408c00), %o5
0xefd9b248:     call  %o5 + 0x3b0       ! 0xee408fb0 <DYNINSTloopTID>
0xefd9b24c:     nop
0xefd9b250:     cmp  %o0, 0
0xefd9b254:     be  0xefd9b264
0xefd9b258:     nop
0xefd9b25c:     b,a   0xefd9b280
0xefd9b260:     nop
0xefd9b264:     b,a   0xefd9b210
0xefd9b268:     nop
0xefd9b26c:     add  %l2, 0x30, %l2
0xefd9b270:     mov  %l2, %o0
0xefd9b274:     mov  %l0, %o1
0xefd9b278:     mov  %l1, %o2
0xefd9b27c:     sethi  %hi(0xee40ac00), %o5
0xefd9b280:     call  %o5 + 0x170       ! <DYNINSTstartThreadTimer_inferiorRPC>
0xefd9b284:     nop
0xefd9b288:     ta  1
0xefd9b28c:     ret
0xefd9b290:     restore
```

Figure 9: Assembly code for inferiorRPC (special trampoline code).

4.5.1 Problems with inferiorRPC

Executing inferiorRPC for threads could case a scenario we call self-deadlock when the thread that performs the inferiorRPC is the thread we intended for (this may seem to be counter intuitive to some people). This happens when the thread has already acquired a lock, and then the inferior-RPC requests the same lock. This causes a deadlock because that these all happens in a single thread of control. To solve this problem we have created an additional thread whose sole existence is to carry out inferiorRPC on behalf of other threads. (Please note that for the current metrics Paradyn supports, and the fact that we only launch an inferiorRPC if the corresponding instrumentation has not been installed, self-deadlock may not occur. The separate RPC thread was introduced for generality.)

4.6 Key Operations

❑ Add Thread:
 Update corresponding Thread Table entry.
 Create same number of vectors of counters and timers as for reminder threads.
 Enable only those counter/timers that apply to the new thread.

❑ Delete Thread:
 De-allocate all counter/timers + all vectors for this thread.
 Update corresponding Thread Table entry.

❑ Add counter/timer:
 Common case: there is space in a vector of counter/timers and we just add this new entry (*all* threads).
 Special case: if there is no space available, then create new vector (for all threads) and add new counter/timer entry.

❑ Delete counter/timer:
 Just tag counter/timer as *invalid*. It does not de-allocate memory.

Some of the advantages and disadvantages of this approach are:

Advantages:
- It allocates less memory than a similar option using an extra level of indirection.
- Execution of mini-tramp code should be fast since counter/timers are accessed directly.

Disadvantages:
- New 2-level memory allocation procedure. Might create memory fragmentation.
- Could be wasting space if computing counter/timer for just one thread but need to add new vector to every thread.
- Only de-allocate memory for counters, timers and vectors when a thread is deleted.

4.7 Timer Issues

Another challenge is to minimize instrumentation cost when measuring time-based metrics on a per-thread basis. Threads are executed by LWPs. To measure the CPU (virtual) time of an individual thread, we must use the per-LWP timer kernel calls and instrument thread context switches to account for thread context switching and migration. To reduce the number of calls to expensive timer routines, we introduced per-thread virtual timers, one for each thread, and implement our performance timers using the virtual timers.

The initial LWP ID of a newly created thread is recorded in a virtual timer and updated at every thread context switch to account for any possible thread migration. We turn a virtual timer off when a thread is de-scheduled (this stops all performance timers for this thread) and turn it back on when a thread resumes execution. Implementing per-thread virtual timers reduces the number of calls to expensive timer routines, and reduces the chance that a timer structure gets accessed by interleaved instrumentation.

4.8 Thread Context Switch

We need to instrument thread context switch in order to properly compute timing metrics. For this, we need to identify the appropriate functions in the thread package. For the Solaris thread package, these functions are: `_resume` (stop timer, thread context switch) and `_resume_ret` (start timer, thread is about to resume execution).

This is a little messy because it requires internal knowledge of the thread package (we actually had to look at the code in order to find what functions to instrument). However, this is only done once per thread package.

4.9 Race conditions

1. *Thread creation*. In order to detect thread creation, If we instrument the routine `_thrp_create` at the exit point (because this is the place when the thread id has been defined). However, at this point the new thread might be already running! (unless is created with the `THR_SUSPENDED` flag on). If this happens, this thread could try to execute instrumentation that is not ready to be executed for this particular thread (the thread table needs to be updated to point to the vector of counter/timers). If this is the case, the routine `DYNINSTthreadPos` will return -2 and we will keep looping until `DYNINSTthreadPos` returns a valid position in the thread table. (We also instrument the routine `_thr_start` which is called before it call the start function of a thread to detect thread creation.)

2. *Thread deletion*. In order to detect when a thread is destroyed, we instrument the routine `_thr_exit_common` at the entry point (it cannot be at the exit point because the current thread is not valid anymore at that point!). The problem with this is that it is possible that since we are saying that a thread has been deleted at the beginning of `_thr_exit_common`, it could still be the case that we try to execute instrumentation for that thread. Of course, the thread still exists for the application, but not for Paradyn. In this particular case, `DYNINSTthreadPos` will return -2 and we will just skip the instrumentation (we can't execute the instrumentation for a thread that has been already deleted). There are cases, however, where we are not computing neither counters nor timers. In these cases, we can execute the instrumentation without problems (e.g. calls to `DYNINSTthreadCreate` or `DYNINSTthreadDelete`).

4.10 New resource: Threads

We add threads under the process hierarchy. In this way in the where axis we would see, for example, that process `java_g{5143_grilled}` has 13 threads labeled in terms of thread id and start function of the thread. However, since resources in the where axis cannot be deleted (at least for now), threads don't disappear from the where axis even if they have been deleted. This might bring some confusion, specially if the user tries to enable a particular metric for a thread that doesn't exist anymore. An example of the new whereAxis can be seen in Figure 10.

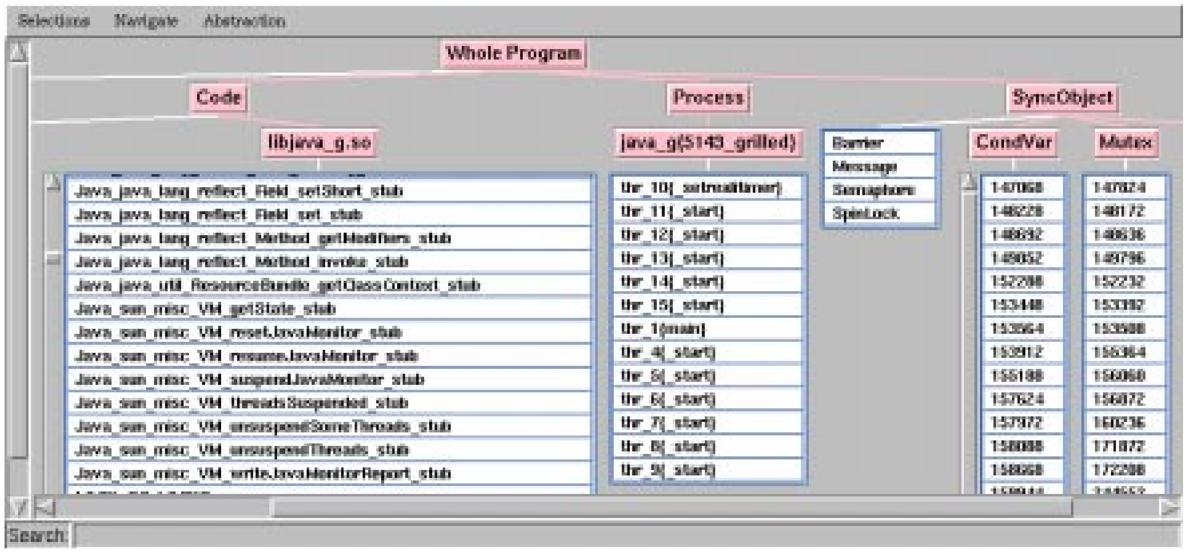
Selections
Navigate
Abstraction
Whole Program
Code
Process
SyncObject
libjava_g.so
java_g{5143_grilled}
Barrier
Message
Semaphore
SpinLock
CondVar
Mutex
Java_java_lang_reflect_Field_setShort_stub
Java_java_lang_reflect_Field_set_stub
Java_java_lang_reflect_Method_getModifiers_stub
Java_java_lang_reflect_Method_invoke_stub
Java_java_util_ResourceBundle_getClassContext_stub
Java_sun_misc_VM_getState_stub
Java_sun_misc_VM_resetJavaMonitor_stub
Java_sun_misc_VM_resumeJavaMonitor_stub
Java_sun_misc_VM_suspendJavaMonitor_stub
Java_sun_misc_VM_threadsSuspended_stub
Java_sun_misc_VM_unsuspendSomeThreads_stub
Java_sun_misc_VM_unsuspendThreads_stub
Java_sun_misc_VM_writeJavaMonitorReport_stub
thr_10{_setrealtimer}
thr_11{_start}
thr_12{_start}
thr_13{_start}
thr_14{_start}
thr_15{_start}
thr_1{main}
thr_4{_start}
thr_5{_start}
thr_6{_start}
thr_7{_start}
thr_8{_start}
thr_9{_start}
147868
148228
148692
149052
152288
153448
153564
153912
155188
157624
157972
158088
158668
147824
148172
148636
149796
152232
153392
153508
155364
156060
156072
160236
171872
172288
Search

Figure 10: Resource Hierarchy

Showing resource hierarchies for code, process and synchronization objects.

4.11 Cost of instrumentation

This section describes the cost of instrumenting threaded code through a few simple benchmarks and compares them with non-threaded Paradyn. Table 1 presents the cost of basic instrumentation primitives, including the cost to execute the base-trampoline, increment a counter, and start and stop a CPU timer. These costs are presented for the threaded and non-threaded Paradyn instrumentations on two different systems: an UltraSPARC II with one 250 MHz processor, and an Enterprise 5000s with twelve 167-MHz processors.

Machine	Base Trampoline		Counter		Start Timer		Stop Timer	
	Non-threaded	Threaded	Non-threaded	Threaded	Non-threaded	Threaded	Non-threaded	Threaded
UltraSPARC II Uniprocessor	125ns	552ns (+342%)	28ns	41ns (+46%)	1.09µs	1.53µs (+40%)	1.15µs	1.47µs (+28%)
Enterprise 5000s	186ns	815ns (+338%)	42ns	65ns (+55%)	1.53µs	2.15µs (+41%)	1.55µs	2.03µs (+31%)

Table 1: Micro Benchmarks

As shown in Table 1, the cost of base-trampoline for threaded Paradyn is about 5 times of that of the non-threaded version. The extra costs are from code added to check for inferior RPCs, to detect already running threads, and to calculate the column address in the Thread Table. Counter primitives for the new version are 50% more expensive than the non-threaded Paradyn, and timer code is about 30%-40% more expensive, mainly because the new version has to go through a level

Machine	**Base Trampoline**		
	Total	**Checking inferior RPC**	**Detecting Threads**
UltraSPARC II Uniprocessor	552ns	119ns (22%)	69ns (12%)
Enterprise 5000s	815ns	186ns (23%)	102ns (13%)

Table 2: Breakdown of Base-trampoline Cost

of indirection. To get an idea of the costs of various components of a base-trampoline, Table 2 lists the cost to check for pending inferior RPCs and cost to detect existing threads.

To get a feeling for the overall cost of the new instrumentation, we instrumented a simple multithreaded application (matrix multiply with 150 lines of C code) and compared it with the cost of instrumenting a sequential version of the same algorithm by the non-threaded Paradyn. Table 1 shows elapsed times of the two versions repeatedly multiplying two 500x500 matrices of floating point numbers. In Table 1, we measure CPU time (inclusive) for the whole program, procedure call frequency and CPU time (inclusive) for the function `innerp`. The procedure call frequency of `innerp` is about 3,500 calls/second on the uniprocessor, and about 10,000 calls/second for the multithreaded version on the multiprocessor. Note that the overhead for the threaded instrumentation is about 2 to 8 times of that for the non-threaded instrumentation. Instrumentation cost is proportional to event frequency. In this example, we instrumented the most frequently called procedure as a stress test.

4.12 Defining metrics for multithreaded programs

Measuring CPU time for the whole program:

Resource list definition (Figure 11). This figure describes the functions that we need to instru-

```
resourceList stopThread is procedure {
items {"_resume", "_thr_exit_common"};
flavor { unix };
library true;
}

resourceList resumeThread is procedure {
items {"_resume_ret"};
flavor { unix };
library true;
}
```

Figure 11: Example of MDL for CPU metric. Resource list definition.

ment in order to catch thread context switches. Whenever `_resume` or `_thr_exit_common` are called, we need to stop the virtual CPU timer for that particular thread. Whenever `_resume_ret` is called, we need to start that virtual timer for that particular thread again.

5 USING THE THREAD-AWARE PARADYN

The thread-aware Paradyn needs the thread-aware Paradyn daemon `paradyndMT` the thread-aware Paradyn runtime library `libdyninstRT_MT.so.1`, and the Paradyn resource file `paradynMT.rc` must be included in the local Paradyn resource file `$HOME/.paradynrc`.

To choose the thread-aware Paradyn runtime library:

```
setenv PARADYN_LIB /p/paradyn/lib/sparc-sun-solaris2.6/libdyninstRT_MT.so.1
```

The following gives an example of the Paradyn PCL file to measuring a threaded application:

```
tunable_constant {
  "costLimit" 20.0;
  "PC_SyncThreshold" 0.0;
  "PC_CPUThreshold" 0.01;
  "PC_IOThreshold" 1.0;
}
exclude "/Code/libintl.so.1";
exclude "/Code/libm.so.1";
exclude "/Code/libw.so.1";
exclude "/Code/libmp.so.1";
process myproc {
        dir "/p/paradyn/applications/Test_dist/threads/matrix";
        command "matrix output";
        daemon mtd;
}
daemon mtd {
      command "/p/paradyn/bin/sparc-sun-solaris2.6/paradyndMT";
      flavor unix;
}
```

6 CURRENT LIMITATIONS AND PROBLEMS

This is a list of the current limitations and problems with the implementation of the instrumentation of threaded applications:

6.1 Infrastructure-level issues

- New flavor for threads `syncWait` metrics.
- LocalAlteration (sparc-solaris). The LocalAlteration code for SPARC-solaris can not relocate some functions correctly, two such examples are libthread `write` and `_exit`

- Atomicity of instrumentation and instrumentability. For large applications, the instrumentation could use up local heap. What is the appropriate action when this happens. The following functionality is desirable:

1. The ability to query the instrumentability of a function.
2. The ability to specifically ask for space within single word jump, or don't care.
3. The ability to instrumentation without doing relocation (when instrumenting a instPoint that can be done without relocation).
4. Delayed instrumentation. Cannot be done the same way as we are doing now for non-threaded applications.

- InferiorRPC: At what level should we provide ways to handle inferiorRPC, e.g., using a LWP or Thread as provided a thread package. How to trigger an inferiorRPC. The current method will always needed to do the initial inferiorRPC.
- Size of Thread Table is now fixed, can we make it extensible?
- Abstraction to deal with different thread packages.
- `exec_time` metric [what is the right semantics of this metric for threads]
- `DYNINSTstartThreadTimer`/`DYNINSTstopThreadTimer`. How to deal with atomicity of the calls in the presence of thread context switch.

6.2 Problems under investigation

- Daemon never receive TRAP due to RPC
- Instrument all thread synchronizations correctly
- `paradyndMT` asserts on `sparc-sun-solaris2.7`
- Spikes of CPU metric/ has to do with inferiorRPC?

6.3 Things that need to be more general

- How to deal with the recycling of thread id. Can our current implementation handle that?
- In the thread-aware Paradyn runtime library, in several places we limit the number of events we can handle, such as number of active threads, number of removed threads, and number of sync objects we can report.
- Eliminate the difference between the thread-aware and thread-unaware daemons.

7 NOTES

1. *Applications*. There are a couple of multithreaded applications for testing in `/p/paradyn/applications/threads`. They are very easy to use and very good for testing (they can create and destroy many threads).
2. *Web sites*. These are some interesting web sites worth looking at:
   `http://wwwwseast2.usec.sun.com/workshop/threads/`
   `http://wwwwseast2.usec.sun.com/workshop/threads/usenix.html`

■