Paradyn Parallel Performance Tools

Developer’s Guide

Release 3.3
January 2002

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

Table Of Contents

Table Of Contents

List of Figures

1 OVERVIEW

This guide is intended to help developers who want to understand the Paradyn source code. It is a rough overview to help with modifications, extensions, and porting efforts. This document is constantly being modified and extended. This document assumes that you are familiar with the fundamentals of Paradyn from the technical papers, manuals, and use of the tool.

We encourage research developments based on Paradyn and hope that this document is of some help. The ultimate source of advice on Paradyn (other than the source code itself!) is the Paradyn development group. Feel free to contact us at `paradyn@cs.wisc.edu`.

1.1 Document revision history

❑ **v3.0**: major revision

- Added new section on Linux-specific implementation: Section 7
- Added new section on retroactive (catchup) instrumentation: Section 5.6
- Added new section on multiple inferior instrumentation heaps: Section 5.7
- Add new section on base-trampoline re-entrancy guards: Section 5.8

❑ **v2.1**: minor revision

- Expanded this overview with summary of new functionality and subsystem dependencies
- Added new section on MDL implementation: Section 9
- Expanded discussion of site dependencies and build configuration in Section 11.2 (and Section 11.5 for Windows NT)
- General update and correction of typos

❑ **v2.0**: major revision

1.2 New functionality for release 3.0

In addition to the new features of summarized in the ***User's Guide*** release notes (Section 1.1), new functionality for Paradyn 3.0 includes:

- ports for x86/Linux and MIPS/Irix
- support for Irix native MPI and MPICH on x86/Linux and x86/Solaris
- multiple inferior instrumentation heaps which support basetramp locality to instrumented functions and consequent atomic single-instruction instrumentation points
- retroactive (catchup) instrumentation
- callgraph-based Performance Consultant exigency search
- instrumentation re-entrancy guards

1.3 New functionality for release 2.1

In addition to the new features of summarized in the ***User's Guide*** release notes (Section 1.1), new functionality for Paradyn 2.1 includes:

- application re-linking requirement removed for SPARC/Solaris
  (Paradyn now dynamically loads its run-time instrumentation library and works with unmodified application executables on SPARC/Solaris and x86/Windows NT)
- automatic code block identification [on Solaris platforms]
  (eliminating the requirement to re-link the application program using explicit code block markers, now also relevant for x86/Solaris). This removes the need for `DYNINSTstartcode` and `DYNINSTendcode` markers which were previously necessary to delimit "interesting" application code, and also leads into the next new feature:
- merged processing of statically and dynamically-linked modules, allowing generalized module and function exclusion [on Solaris platforms].
  In general, the code for handling statically and dynamically linked code on Solaris has unified. This unification removes the requirement of re-linking target application programs with the `DYNINSTstartcode` and `DYNINSTendcode` markers. It also generally increases the amount of application code which is instrumented, and necessitates explicitly excluding more modules in the Paradyn configuration files.
- handling of stripped dynamic libraries [under Solaris]
  The run-time linker's dynamic symbol table (`.dynsym`) is now parsed allowing instrumentation of stripped shared objects and stripped dynamic executable files on SPARC&x86/Solaris.
- 2-pass function relocation and expansion [on SPARC architecture]
  Previously, when relocating a function which could not be instrumented in place, Paradyn made a single pass over the function machine code and patched targets for machine instructions which specify a code address during this single pass. This meant that instructions which specified a destination address inside the same function (such as branch instructions) did not have correct targets if any extra code needed to be inserted between the original location and the target address. Function relocation is now done in two passes: the first pass detects locations at which extra instructions will need to be inserted, and the second pass relocates the function, patching address targets inside of the function accrordingly. This feature is only currently implemented on the SPARC architecture.
- better handling of optimized code [on SPARC architecture].
  A number of sequences which appear in heavily optimized SPARC code are correctly parsed and instrumented by Paradyn version 2.1 which were not correctly handled by the Paradyn 2.0 release. To date these code sequences have only been found in heavily optimized system libraries (especially **libc**).
- more powerful, simplified MDL syntax for metric definition
- enhanced metrics for I/O in MPI programs [on the SP2]
- scalability to monitor larger numbers of processes
- easier, parameterized source build (with PVM support now a build option)

1.4 Paradyn subsystems and source code structure

Paradyn consists of several subsystems which are listed in Figure 1.

`paradyn`	Paradyn front-end.
`paradynd`	Paradyn daemon.
`dyninstAPI`	A separate library for dynamic instrumentation, also used as part of the Paradyn daemon.
`dyninstAPI_RT`	dyninstAPI run-time instrumentation library (not used by Paradyn).
`rtinst`	Paradyn run-time instrumentation library.
`visiClients`	Visualization client programs for performance data: rthist (run-time histogram), table, barchart, phaseTable, terrain and tclVisi. Also the termWin program for output data of application processes
`util`	Utility functions used by other sub-systems.
`thread`	General purpose custom multithreading package.
`igen`	RPC interface generator.
`visi`	Visi interface library.

Figure 1: Paradyn (and dyninstAPI) subsystems.

"Paradyn" is the front-end control process which typically runs on desk-top workstations. "Paradynd" is the Paradyn daemon process that run on each host on which you run your application program. "dyninstAPI" contains the code of the dynamic instrumentation application program interface, or *dyninstAPI*[1] . In a future release, the "dyninstAPI" directory will contain just the dyninstAPI library but today it is an intermediate stage in the separation of that functionality from the Paradyn daemon. "rtinst" is the source for the library `libdyninstRT` that is linked into each application program to support Paradyn's dynamic run-time instrumentation. "visiClients" are separate run-time visualization programs that can be started by Paradyn to display performance data. The remaining items ("util", "thread", "igen" and "visi") are libraries and utilities used by parts of Paradyn and dyninstAPI: component dependencies are summarized in Figure 2.

	Core subdirectory	Component dependencies			
		Util	*Visi*	*Igen*	*Other*
Basic Components:					
`libpdutil`	util				

Figure 2: Paradyn/dyninstAPI module structure and dependencies.
Libraries and associated include files are common module dependencies, often supplemented with process interface routines generated by Igen from interface specifications. Occasionally, direct source sharing is also employed (e.g., between the dyninstAPI and paradynd).

1. also available from `http://www.cs.umd.edu/projects/dyninstAPI`

`libpdthread`	thread				
`libvisi`	visi	h		exe	
`igen*`	igen	h&lib			
dyninstAPI:					
`libdyninstAPI`	dyninstAPI	h			dyninstAPI_RT
`libdyninstAPI_RT`	dyninstAPI_RT	h			
Key Subsystems:					
`libdyninstRT`	rtinst	h			
`paradynd*`	paradynd	h&lib		exe	paradyn, dyninstAPI, rtinst
`paradyn*`	paradyn	h&lib	h/.I	exe	(libpd)thread, paradynd
Visualizers:					
`barChart*`	visiClients/barchart	h&lib	h&lib		paradyn
`phaseTable*`	visiClients/phaseTable	h&lib	h&lib		paradyn
`rthist*`	visiClients/histVisi	h&lib	h&lib		paradyn
`tableVisi*`	visiClients/tableVisi	h&lib	h&lib		paradyn
`tclVisi*`	visiClients/tclVisi	h&lib	h&lib		paradyn
`terrain*`	visiClients/terrain	lib	h&lib		[paradyn]

Figure 2: Paradyn/dyninstAPI module structure and dependencies.
Libraries and associated include files are common module dependencies, often supplemented with process interface routines generated by Igen from interface specifications. Occasionally, direct source sharing is also employed (e.g., between the dyninstAPI and paradynd).

Note: the terrain visi has not yet been ported to Windows NT. While the same source structure applies, as described in the discussion which follows, differences are dealt with in Section 11.5

The root of the Paradyn source code tree has one directory for each one of these modules. Each module directory is divided into several sub-directories:

`h`: this directory contains the exported interface of the module, usually C or C++ header files, or Igen interface specifications (files with suffix `.I`).

`src`: this directory contains the source code for the module and header files that are not part of the exported interface.

compilation directories (`<arch>-<vendor>-<os>`, as provided by *sysname* from the GNU configuration system, one for each supported platform): These directories contain a Makefile and machine derived files that are built as part of the compilation process, such as intermediate files generated by Igen, flex, and bison, and object files.

Each module directory also contains a configuration file (`make.module.tmpl`) that is included by the Makefile in the compilation directories.

The root directory of the source code tree also contains a Makefile, which can be used to build all of the components of the system, and three configuration files that are included by the Makefiles in the compilation directories of each module:

`make.config`: general definitions for all Paradyn modules, such as compilers and other programs to use, flags, search path for include files, libraries, etc. This file generally needs to be updated for each installation, with the desired configuration options, valid paths to the programs, and libraries; see Section 11 for further details.

`make.library.tmpl`: general definitions for modules that generate libraries.

`make.program.tmpl`: general definitions for modules that generate programs.

The build also uses a shell/command script, `buildstamp`, provided in the scripts directory (which also includes a copy of `sysname`).

A more complete description of the configuration and Makefiles used in Paradyn appears in Section 11.

2 PARADYN PACKAGE DEPENDENCIES

This section lists the packages needed to build Paradyn on Unix systems: some Windows NT differences are mentioned here, but see Section 11.5 for details. For each package, we list where in the Paradyn source code the package is needed, the version of the package currently used, how to get the package, and some additional information. If you notice any packages that we have missed listing below, please let us know.

❑ **gcc/g++:**

- *Where used:* compiling all of Paradyn.
- *Version:* gcc-2.95. We currently build using 2.95.3 for all platforms; Paradyn may compile with gcc-2.96 (the so-called “Red Hat gcc”), or with gcc-3.0.1, but it will be unable to instrument binaries created with these compilers.
- *How to get:* ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.tar.gz (or other suitable version).
- *Comments:* close to impossible to work without a good C++ compiler. We use some non-standard features (such as long long), which may not be supported by other compilers.
- *Windows NT:* Visual C++ 6.0 is used instead. Compiling Paradyn with gcc/g++ is still untested on this platform.

❑ **GNU make:**

- *Where used:* building all modules in Paradyn
- *Version:* currently using make-3.74
- *How to get:* http://wuarchive.wustl.edu/systems/gnu/make-3.74.tar.gz
- *Comments:* we use includes, conditional defines, and other features specific to GNU-make.
- *Windows NT:* ***nmake*** is used instead, which has a different syntax and capabilities, necessitating a separate set of make configuration files called `nmake.config`, `nmake.*.tmpl.` These configuration files may be deleted if you’re not working with Windows NT).

❑ **Perl5:**

- *Where used:* in the ‘tcl2c’ script to convert Paradyn Tcl files to C++; also used (though this could be easily changed) in `make.config`
- *Version:* perl5.xxx
- *How to get:* http://mox.perl.com/ and explore, or http://wuarchive.wustl.edu/systems/gnu/perl5.002.tar.gz
- *Comments:* possible to rewrite tcl2c in almost any language.
- *Windows NT:* currently not needed on this platform.

❑ **Tcl/Tk:**

- *Where used:* user-interface of Paradyn, tclVisi package, barChart, tableVisi, etc.
- *Version:* Tcl-8.3.3, Tk-8.3.3 (or higher).
- *How to get:* http://tcl.activestate.com and explore.
- *Comments:* Tcl/Tk enables greater portability for Paradyn's user interface.
- *Windows NT:* we recommend using the pre-built binary Tcl/Tk package from tcl.activestate.com on Windows systems.

❑ **Xaw, Xext, Xt:**

- *Where used:* 3D terrain visi.
- *Version:* Xaw-5.0, Xext-4.10, Xt-4.10 (or higher).
- *How to get:* http://www.x.org/ and explore.
- *Comments:* other versions may require re-compiling rthist.
- *Windows NT:* not used on this platform.

❑ **Bison, Flex:**

- *Where used:* Igen, MDL.
- *Version:* bison v1.24, flex 2.5.2 (or higher).
- *How to get:* http://wuarchive.wustl.edu/systems/gnu/bison-1.24.tar.gz and flex-2.5.2.tar.gz
- *Windows NT:* these are needed to build `paradynd`, and you can either build them from the sources or look around for pre-built versions for Windows NT.

❑ **libelf: (Linux only)**

- *Where used:* paradyn daemon, dyninstAPI.
- *Version:* Use a version appropriate for your kernel version. For example, Red Hat 6.2 users can use libelf-0.6.4-4.i386.rpm, Red Hat 7.1 users can use libelf-0.6.4-7.i386.rpm.
- *How to get:* An RPM is included in Red Hat distributions.
- *Comments:* This package contains the libelf library and headers, used by Paradyn's daemon and dyninstAPI to access ELF files under Linux.

❑ **ONC RPC: (Windows NT only)**

- *Where used:* Paradyn daemon, dyninstAPI, libpdutil
- *Version:* v1.10 or later
- *How to get:* ftp://grilled.cs.wisc.edu/~paradyn/etc/oncrpc112winnt.tar.gz
- *Comments:* the ONC RPC implementation of Sun RPC for Windows NT originates from Martin F. Gergeleit (http://set.gmd.de/~mfg/oncrpc.html), however, the file `RPC/XDR.H` needs to be exchanged with the one in the Paradyn release to compile successfully with Visual C++ 6.0.

❑ **rshd:**

- *Windows NT:* if you wish Paradyn to be able to automatically start applications and Paradyn daemon processes on remote Windows NT systems, an *rsh* daemon process, such as

WRSHDNT, is required to be running on the remote system.

3 PARADYN FRONT-END

The Paradyn front-end is a multi-threaded system that consists of several modules: the data manager, the user interface, the visualization manager, and the Performance Consultant. Each of these modules is a separate thread. The Paradyn process starts by creating each module's thread, and invoking initialization routines for each thread. After each thread is initialized, the commands in the Paradyn configuration files are processed, and control is passed to the threads.

The User-Interface thread (UI) is responsible for receiving user's commands and managing the display windows (the Paradyn Main Console Window, the Where Axis, and the Performance Consultant Window). The Data Manager thread (DM) is responsible for handling requests from other threads for data collection, for receiving performance data from the Paradyn daemons and delivering them to the requesting threads, and for managing information about phase, metrics, and the resource hierarchy. The Performance Consultant thread (PC) is responsible for the automated search for performance bottlenecks in the application. The Visi Manager thread (VM) is responsible for managing visualization processes (like the run-time histogram and barchart processes) and for communication between each visualization process and the Data Manager.

The source code for the Paradyn process is divided in several directories, including one directory for each thread: `DMthread`, `PCthread`, `UIthread`, and `VMthread`. There is also a directory called `TCthread`, which has code to handle tunable constants. (Tunable constants are not a thread, however, they are managed by the UI thread). The `met` directory contains the parser for the Paradyn Configuration Language; the `VISIthread` directory contains the code for visi threads, which are created by the VMthread when a new visualization process is started; the `pdMain` directory contains the Paradyn main routine.

The following sections describe the major modules of the Paradyn front-end.

3.1 Data Manager

The Data Manager (DM) is one of the threads of the Paradyn main process. The Data Manager handles requests from other threads for data collection, delivers performance data from the Paradyn daemon(s) to the requesting thread(s), and maintains and distributes information about the metrics and resource hierarchies for the currently defined application.

Performance data collection

The Data Manager handles requests from other threads for performance data collection. For this purpose, the DM provides the "public" procedure `dataManager::enableDataRequest` (`DMpublic.C`). This procedure will receive, among other parameters, the metric/focus pair we want to enable, the perfStreamHandle of the calling thread, the identifier of the phase for which data is requested, and other necessary information. This procedure will then call the corresponding procedures to enable the data collection process in the Paradyn daemon(s).

In general, all the requests to the DM from other threads, are handled in the file `DMpublic.C`.

Performance data delivery from the Paradyn daemon(s)

Once the data has been successfully enabled, the Paradyn daemon(s) will start sending data to the requesting threads through the DM. The DM will receive trace records and send them to the requesting thread (`DMperfstream.C`).

Metrics and resource hierarchies management

There are objects that can be created and destroyed and the DM has to notify the corresponding threads about all these changes. If a new resource is created, for example a new process, then the Paradyn daemon will make a "call back" to the DM, and then the DM will notify the corresponding threads (e.g. the UI). Call backs are defined in the file `DMmain.C`.

DM objects

The major objects used in the DM thread are described below. The file in which the class of each object is defined is given in parenthesis.

These objects, once created, are never destroyed:

`resource` (`DMresource.h`): the static items basically manage a "database" of all resources. The non-static items gives you information about a single resource.

`resourceList` (`DMresource.h`): a "list" of all resources in the system.

`metric` (`DMmetric.h`): contains all the information related to a metric (e.g. name, units, type, etc).

`phaseInfo` (`DMphase.h`): information about phases in the system.

These objects, once created, can be destroyed:

`metricInstance` (`DMmetric.h`): this class contains information about the particular "instances" of all metrics created during the execution of the application being analyzed. (If a metric is "enabled", we are creating a new metric instance; if the same metric is "disabled", we are destroying it).

`performanceStream` (`DMperfstream.h`): the `performanceStream` class is basically a consumer of performance data. Its main function is to provide the means to receive data from the Paradyn daemon(s) and send it to the requesting threads.

`paradynDaemon` (`DMdaemon.h`): a handle to a running Paradyn daemon (`paradynd`). This class provides method functions for process and daemon control as well as for enabling and disabling data collection. At this moment, if a particular paradynd is removed (e.g., exits), then Paradyn has to exit too. In other words, we can't destroy a Paradyn daemon in the current implementation.

All DM objects should be referred to by their handles outside of the DM thread. The only operation that clients should perform with DM handles is equality testing (this operation will always be supported by DM handle types, so clients can compare handle values directly), any other information that a client needs about a DM object can be obtained by passing the appropriate handle to a `dataManager` interface routine.

Within the DM thread, care should be taken when using pointers to objects that are not persistent (`metricInstance` and `performanceStream`)

DM handles are not reused over the execution of Paradyn, but `metricInstance` and `performanceStream` handles may be invalid. For example, enabling a metric/focus pair, disabling it, and then re-enabling it may result in two different metricInstance handles to be associated with the pair.

3.2 Visi Manager

The Visi Manager is a thread in the Paradyn process. It contains information about the visualizations in the system, and it accepts requests from other threads to start or to kill visualization processes. When the visi manager receives a request to start a new visualization, it creates a visi thread. The visi thread then starts the external visualization process, and acts as an interface between the visualization process and the Paradyn process.

Visi Manager types

The following is a description of the types used by the visi thread (these types are defined in `VMtypes.h`):

`VMvisis`: The visi manager keeps a vector of `VMvisis` elements. Each element in the vector contains information about a visualization that has been added to Paradyn. The visi manager uses this information to start the visualization process. (Note: the `matrix` and `numMatrices` elements are not currently used.)

`VMactiveVisi`: The visi manager keeps a vector of `VMactiveVisi` elements. There is one element in this vector for each visualization process that is currently executing. When a new visualization process is started, a new element is added to this vector, and when a visualization process exits, its corresponding element is removed. The visi manager uses the information in each element to communicate with the visualization process. Each element contains information about the type of visualization that is running, and about the visi thread that is associated with the visualization process.

`visi_thread_args`: This struct is used when the visi manager thread creates a visi thread. It contains information that the visi thread needs to start the visualization process. (Note: the matrix element is not currently used.)

Visi Manager interface routines

Visi manager interface routines provide information about visualizations in the system, and provide a mechanism to control visualization process creation and deletion. These routines are defined in `VM.I`.

3.3 Visi threads

Visi threads are the only threads that are not persistent over the execution of the Paradyn process. There may exist zero, one, or more instances of a visi thread at any time. They are the only threads that can be created and destroyed at any point in Paradyn's execution.

A visi thread is created by the visi manager thread when it receives a request to start a visualization process. The visi thread starts the external visualization process, and acts as an interface

`VMActiveVisis`	Returns a vector of information about all visualization processes currently running.
`VMAvailableVisis`	Returns a vector of information about all the different visualizations that are part of Paradyn.
`VMAddNewVisualization`	Takes information about a visualization process, and adds it to its list of `VMvisis` elements.
`VMCreateVisi`	Starts a visualization process.
`VMDestroyVisi`	Kills a visualization process.
`VMVisiDied`	Called by a visi thread when its associated visualization process has exited. It cleans up any state that the visi manager thread has been keeping for this process.

Figure 3: Visi Manager interface

between Paradyn and the visualization. There is one visi thread for every visualization process that is executing. The visi thread receives requests from the visualization to change its set of performance data, and forwards these requests to other threads in the Paradyn process. From the other threads, the visi thread receives performance data and meta data that it packages and forwards to the visualization process.

Because there can be multiple instances of a visi thread, visi threads must use thread local data to keep any unique information that they need to interact with their associated visualization process.

Visi thread types

Types used by the visi thread are defined in `VISIthreadTypes.h`. Each visi thread has an element of type `VISIthreadGlobals` in its local data. This element contains state information about the visualization process it is associated with, and about the other threads with which it needs to communicate. Figure 4 provides a description of this struct.

The Visi thread and the Visi interface

Each visi thread is a client instance of the visi interface, and each visualization process is a server instance of the visi interface. The visi interface is defined in `visi.h`. The visi server routines are implemented in Paradyn's visualization library (visiLib). This library is then linked with visualizations that want to receive Paradyn performance data. For a complete description of visiLib see the ***Paradyn Visi Programmer's Guide***.

The visi thread implements the visi interface client routines. These are upcalls that are made by the visualization process to the Paradyn process, and they provide a mechanism for a visualization process to subscribe or un-subscribe to performance data, or to start a new phase. When the visi thread receives an upcall from a visualization process, it typically makes one or more calls to other threads in the Paradyn process to satisfy the visualization's request.

Field	Use
`ump`	Used to call user interface RPCs.
`vmp`	Used to call visi manager RPCs.
`dmp`	Used to call data manager RPCs.
`visip`	Used to communicate with the visualization process.
`ps_handle`	Used as an identifier by the data manager, data manager calls and callbacks typically have a `perfStreamHandle` argument.
`fd`	File descriptor used to communicate with the visualization process.
`buffer`	A buffer of performance data (the visi thread sends data to the visualization process a buffer full at a time).
`quit`	Flag that tells the visi thread to exit.
`start_up`	Flag that tells the visi thread that there is some initialization that it needs to do.
`bucketWidth`	Bucket width associated with the data buckets that are being sent by the data manager to the visualization process.
`currPhaseHandle`	Handle for the current phase.
`args`	Arguments used to start the visualization process.
`mrlist`	List of metric/focus pairs that the visualization process is currently subscribed to.
`request`, `retryList`, `numEnabled...`	Stores information about any outstanding enable requests that have been made by the visualization.

Figure 4: VISIthreadGlobals struct members.

The Visi thread and the Data Manager

The visi thread makes data enable, and disable requests to the data manager thread on behalf of the visualization process. A data enable request is asynchronous, so the visi thread must keep state about the request until it receives an asynchronous upcall from the data manager with the response. Once the visualization has subscribed to some performance data, the data manager thread will send this data to the visualization's visi thread. The visi thread packages the data and sends it to the visualization process.

The visi thread is a data manager client thread, and thus implements data manager client routines. Since there are other data manager client threads in the Paradyn process, each thread contains code that implements its version of the data manager client thread routine, and then it registers this routine as a callback with the data manager thread. When the data manager makes an upcall to a data manager client thread, the client thread's callback routine is called.

To communicate with the data manager, the visi thread must first create a performance stream. When the visi thread makes a request to create a performance stream it also registers all its call-back functions with the data manager. The data manager returns a performance stream handle that is used in all subsequent communications between the visi thread and the data manager.

Interface routines

The visi thread acts more as a client thread in the Paradyn process, and thus only has one server routine defined in `VISIthread.I`:

`VISIKillVisi`: called by the VM thread when a request is made to kill the visualization process.

The file `VISIthreadmain.C` contains the VISIthread main loop, and callback routines for UI, and DM upcalls.

The file `VISIthreadpublic.C` contains VISIthread server routines, and visi interface upcalls.

3.4 User Interface (UI) thread

The user interface (UI) thread handles all graphical displays in Paradyn. It has several tasks to perform, including the Where Axis Window, the Tunable Constants Window, the Paradyn Main Console Window, the Performance Consultant Window, the Error Dialog Window, the Call Graph Window, etc. For the most part, these tasks are handled via the Tcl/Tk package. Simultaneously, however, the UI thread must listen for Igen messages from the data manager thread; the most numerous being "new-resource" messages, which require the UI thread to add items to the Where Axis display.

UI main loop

`UImain()` of file `UImain.C` is the entry point to the UI thread. After creating a number of tunable constants, it calls `initialize_tcl_sources()` to read in Paradyn's Tcl code. The source (`.tcl` files) for such code is in the `paradyn/tcl` directory. When compiling Paradyn, the "tcl2c" script converts the `.tcl` files into a `tcl2c.C` file, which contains a function `initialize_tcl_sources()`. Calling this function (as UImain does now) reads in all of our Tcl scripts. For this reason, the `.tcl` files do not need to be distributed in a binary release of Paradyn.

`UImain()` soon calls `msg_bind()` on `XConnectionNumber()` of the X display. In this way, we can wait for X events. X provides a number of functions (such as `XNextEvent()`) to do this more cleanly, but since the UI thread needs to wait not just on X events but also for Igen messages, this roundabout approach is needed.

The main UI loop is as follows. The routine `processPendingTkEventsNoBlock()` is called to process any pending X events (i.e., any Tcl/Tk graphical events) without waiting. Then, we call libthread's `msg_poll()`, which will wait for either an Igen message, an X event, or a keyboard event (previous calls to `msg_bind()` determines what `msg_poll` waits for). We then determine which of the 3 events occurred, and process the event accordingly. For X events we call `processPendingTkEventsNoBlock()`; for keyboard events, we call `StdinProc()`; for Igen events we call the appropriate Igen waitLoop() routine. `processPendingTkEventsNoBlock()`

simply calls `Tk_DoOneEvent()` until no more Tk events are pending. In this way, we handle mouse clicks, etc., in all of Paradyn's windows.

Where Axis

In `paradyn/src/UIthread`, files dealing with the where axis are `whereAxis.h` and `.C`, `where4tree.h` and `.C`, `whereAxisTcl.h` and `.C`, `where4treeConstants.h` and `.C`, `rootNode.h` and `.C`, and `abstractions.h` and `.C`. Miscellaneous graphical routines are supplied in `scroll-bar.h` and `.C` and `tkTools.h` and `.C`. Classes helping calculate exactly which node was clicked on are in `simpSeq.h` and `.C` and `graphicalPath.h` and `.C`.

Class `abstractions` (`abstractions.h` and `.C`) holds all of the where axes, and also maintains variables to manage the Tk window. Method `add()` is called when a new where axis (a new abstraction) is created. `getCurrent()` returns the current where axis structure. `getCurrAbstractionSelections()` returns the set of resources selected. Class `whereAxis` (`whereAxis.h` and `.C`) holds information on a single where axis. Variable `rootPtr` is the root node of this where axis. Class `where4tree` (`where4tree.h` and `.C`) holds information on a single node in the where axis. Member `theChildren` holds the vector of children of this node. `addChild()` is called when a new child is created. `draw()` draws the node and recursively draws the children. Method `draw_listbox()` draws a node's listbox; method `scroll_listbox()` handles scrolling it. Class `rootNode` (`rootNode.h` and `.C`) defines the input class to the template class `where4tree<>`. File `whereAxis.tcl` contains the part of the where axis code that is written in the Tcl/Tk language. It mainly concerns the frame of the window and its menus. The body of the where axis is drawn in C++ code using a combination of calls to internal Tk C language routines and Xlib routines (for speed).

Performance Consultant window (Search History Graph)

In `paradyn/src/UIthread`, files dealing with the Performance Consultant display are `shgPhases.h` and `shgPhases.C`, `shg.h` and `shg.C`, `shgRootNode.h` and `shgRootNode.C`, `shgTcl.h` and `shgTcl.C`, and `shgConsts.h` and `shgConsts.C`. Files shared with the where axis are `where4tree.h` and `where4tree.C` as well as helper classes provided in `scrollbar.h` and `scrollbar.C`, `tkTools.h` and `tkTools.C`, `simpSeq.h` and `simpSeq.C`, and `graphicalPath.h` and `graphicalPath.C`. `shgPhases.h` and `shgPhases.C` provide class `shgPhases`, which manages the collection of search history graphs (one per phase). Method `change()` switches displays; `draw()` draws the current search history graph; `addNode()` adds a node to the current graph; `addEdge()` connects a node to its parent; and, `configNode()` changes a node's semantics (i.e. true, false, unknown, etc.). `shg.h` and `shg.C` provide class `shg`, which manages a single search history graph. There are many internal similarities to the `whereAxis` class. `rootPtr` holds the root node of this shg. `draw()` draws the shg. `addNode()` adds a node to the shg; `configNode()` changes a node's semantic meaning; `addEdge()` connects a node to its parent. `where4tree.h` and `where4tree.C` manage an individual node of class `shg`; it was discussed above in the where axis. `shgRootNode.h` and `shgRootNode.C` manage class `shgRootNode`, the template input parameter to class `where4tree<>`. File `shg.tcl` constrains the part of the Performance Consultant window written in the Tcl/Tk language. It mainly concerns the frame of the window and its menus. As with the where axis, the shg itself in the center of the window is drawn entirely with calls to internal Tk C routines or Xlib routines, for speed.

Tunable constants

The tunable constants dialog is managed in `tclTunable.tcl` in `paradyn/tcl`. Routine `tunableIntialize()` sets things up; routine `processShowTunableDescriptions()` creates the Tunable Descriptions dialog.

`tclTunable.h` and `tclTunable.C` (in `paradyn/src/UIthread`) provide the implementation of a "tclTunable" command that is called from the above `.tcl` files to gain access to the internal tunable constants database.

The internal tunable constants database is maintained files `tunableConst.h` and `tunableConst.C` (in `paradyn/src/TCthread`).

Status lines

The status lines (which appear in the Paradyn main console window) are managed internally by `Status.h` and `Status.C` (in `paradyn/src/UIthread`). Some of the code to manage the status lines is written in Tcl/Tk; file `status.tcl` (in `paradyn/tcl`) has that code. Status lines for nodes/processes are distinguished from generic Paradyn and application status lines, appearing in a separate resizable and scrollable area of the console window.

Paradyn Main Control window

Most of the Paradyn main window is managed by Tcl/Tk code. File `mainMenu.tcl` (in `paradyn/tcl`) creates the window, its menus, etc. Routines in `shg.tcl`, `whereAxis.tcl`, `tclTunable.tcl` are invoked when the Performance Consultant, Where Axis, and Tunable Constants, respectively, are chosen from the main window's menu. These files have been discussed previously. `startVisi.tcl` is invoked when "Start A Visi" is chosen from the main window's menu. `mets.tcl` is invoked when Paradyn needs a metric selection from the user (in response to a visualization add request). `applic.tcl` maintains the dialog box for starting a new application.

3.5 Performance Consultant thread

The Performance Consultant (PC) thread conducts an automated search for performance bottlenecks. One search may be conducted per phase, for a maximum of two simultaneous searches (one global, one current). The Performance Consultant thread interacts with the DM thread to enable/disable metric/focus pairs and for information about resources, and interacts with the UI thread to control the content of the Performance Consultant window. The Performance Consultant may be viewed as a stream of incoming data, a set of experiment definitions, and a search control strategy for starting and halting individual experiments.

The data stream

Data is obtained by making instrumentation enable requests of the daemon via the data manager. The incoming stream of data is handled by a series of filters. A filter is defined by two base classes, dataProvider and dataSubscriber. There are three types of filters in the Performance Consultant:

`PCfilter (dataProvider):`
in: raw data manager data for a single metric/focus pair,
out: average metric/focus values for uniform time intervals,
subscribers: one or more `PCmetricInst`s.

`PCmetricInst (dataSubscriber, dataProvider):`
in: `PCfilter` output for uniform time intervals for a set of metric/focus pairs,
out: computed from data plus specified arithmetic operator, for a particular time interval,
subscribers: one or more experiments.

`experiment (dataSubscriber):`
in: `PCmetricInst` output (a single value),
out: `changeConclusion`, `changeTruth` calls to the search node,
subscribers: none.

Experiment definition

A `PCmetric` is a set of data manager metrics plus an arithmetic operation (currently +, -, *, /, max). A hypothesis is a specification of a condition to test for plus the data and computation necessary to perform the test. The computation is specified as a `PCmetric` plus a threshold. An experiment is defined by a hypothesis plus a particular focus. Using the hypothesis definition, the appropriate metric/focus pairs are enabled for the `PCmetric`; once data starts flowing from the data manager the resulting value is periodically compared to the threshold. The set of hypotheses is hierarchical and is referred to as the Why Axis.

Search control

All data structures for one search are gathered in an instance of `PCsearch`: `PCmetricInstServer` is the data source; `searchHistoryGraph` is a DAG which contains all info about the tests performed; and two static `PriorityQueue`s, one global and one current, hold all ready search nodes. The total cost of instrumentation is controlled by three thresholds: a cost limit, the total number of active experiments, and the total number of pending enable requests.

Starting up a particular experiment

1. Get estimated cost: when a node is expanded, a request is made to the Data Manager for the predicted cost for each new child node; pointers to the new PCmetric filters are stored on a waiting list `costServer::costRecords`. When the cost is received from the Data Manager, the record is retrieved, and method `updateEstimatedCost()` is invoked for the appropriate PCmetric filter. The PCMetric filter notifies the experiment, which invokes `searchHistoryNode::estimatedCostNotification()`. The `shn` routine places the node

onto the PC run queue.

2. Enable request(s): when a node is launched from the PC run queue, one or more enable requests are made to the Data Manager for the metric/focus pairs used by that experiment. None to all of these pairs may already be enabled, in which case the existing data filter is subscribed to and no new request goes to the Data Manager. As each response comes back from the Data Manager, the PCmetric filter is notified; when all required data is enabled, the experiment is notified and the node display is changed to active.
3. Change to true: when a node's status changes from unknown to true, both parent and children may be affected. If the parent is virtual, its truth value is just the OR of its children's, so its truth value may change. If the node has not been expanded, it is so at this time, and estimated cost is requested for each child (step 1 above). If the node has been expanded in the past, then the child nodes will already have an estimated cost; they are added back to the run queue to await step 2 above. In most cases a change from false to true is not possible, since nodes are deactivated when they become false: this can happen, however, if the node is persistent or if the node's parent changes.
4. Change to false: when a node's status changes from unknown to false it is deactivated and not expanded. If it changes from true to false then it must be deactivated, plus its parent(s) and children must be notified. Every node but the root must have at least one true parent to remain active, so notifying the children generally results in deactivating them.

4 VISI LIBRARY

VisiLib is a library and remote procedure call interface for accessing Paradyn performance data in real-time. VisiLib provides an open interface to Paradyn data, and allows a programmer to build external visualization processes (*Visi*s). All performance visualizations in Paradyn are implemented as visis. The visi programmer uses the interface defined in `visualization.h` to access performance data. VisiLib uses the Igen interface that is defined in `visi.h` to communicate with Paradyn. `visualization.C` contains the implementation of routines defined in both these header files. VisiLib also defines a type (`DataGrid`) that is the visualization's interface to performance data. A complete description of VisiLib can be found in the ***Paradyn Visi Programer's Guide***.

5 PARADYN DAEMON

The Paradyn daemon (`paradynd`) is the back-end of the Paradyn tool. When running a parallel program (such as MPI), there will be several daemons running at the same time, one on each node. Each `paradynd` communicates, using Igen RPC calls, with the Paradyn front-end. There is no direct communication between the Paradyn daemons (except in the case where a daemon is responsible for starting other daemons).

5.1 Introduction

Paradyn daemons have several responsibilities:

1. Starting and controlling the execution of application processes.
2. Reading the application's symbol table.

3. Reading the application's binary image to find instrumentation points.
4. Evaluating metrics, generating code, and inserting instrumentation into application processes.
5. Periodically sampling performance data from the application and forwarding values to the Paradyn front-end (Section 5.5).

Daemons are started by the Paradyn front-end using `rsh` or `rexec` (when the Paradyn front-end runs on a different machine/node than the application) or `fork`/`exec` (when the Paradyn front-end runs on the same machine/node as the application; Windows NT uses `CreateProcess`). The front-end passes the flavor of the daemon (e.g. PVM, MPI, etc.), the name of the machine where the front-end is running and socket address for connection as command line arguments to the daemon. The daemon then connects to the front-end. When PVM is being used, only one daemon is started by the front-end. This daemon then uses `pvm_spawn` to start the other daemons on all nodes of the PVM virtual machine. (The code to parse arguments and connect to the front-end is in `main.C`, and the code to start spawning other Paradyn daemons with PVM is in `pvm_support.C`.)

The interface between the `paradynd` processes and the Paradyn front-end is defined in file `paradyn/h/dyninstRPC.h`. In most cases the `paradynd` acts as a server, receiving requests from the Paradyn front-end, but there are also many upcalls from the `paradynd` to the front-end. Most RPC calls defined in the interface are implemented in `dynrpc.C`, where calls to other modules of the `paradynd` process are made as appropriate.

Daemons start application processes using `fork/exec` (Windows NT uses `CreateProcess`). Daemons use `ptrace` or `/proc` file system calls to insert instrumentation into the application processes (Windows NT uses `ReadProcessMemory` and `WriteProcessMemory`). The standard output and error messages of the application and Paradyn daemon are redirected to a Tcl/Tk front-end terminal window. Output from Paradyn daemon is displayed in a different color from that of the application.

The function `controllerMainLoop()` (defined in `perfStream.C`) is the main loop of the `paradynd`. At each iteration of this loop, the daemon checks for data coming from the application processes through the pipes, for requests by the front-end, and for signals received by the application processes.

Before going into its main loop, each daemon received metric definitions from the Paradyn front-end. The representation of the metrics is provided in the `paradyn/h/dyninstRPC.h` file.

5.2 Application processes

The class `process` (defined in `process.h`/`process.C`) provides a representation for application processes. It provides machine independent abstractions for creating new process, running, stopping, reading, writing, and intercepting signals of application processes.

Several methods of the class process have platform-dependent implementations, in the form of `ptrace` calls or `ioctl` calls to the `/proc` file system. This platform-dependent functions are implemented in the operating system specific files (`e.g. solaris.C`, `aix.C`).

The class `inferiorHeap`, also defined in `process.h/process.C`, provides a representation for the inferior heap in the application process, and functions for allocating and de-allocating memory blocks. The inferior heap is a block of memory in the application process address space where the daemon writes instrumentation code. In addition, on platforms *not* supporting shared-memory data sampling (Section 5.5), the application also stores its counters and timers here.

5.3 Object file processing

The Paradyn daemon reads the object file of an application process to find the symbols (functions, modules, and global data) and instrumentation points. The class `image` (defined in `symtab.h`) provides a representation for the application's object image. The first step in the processing of the object file is to read the `a.out` format file and obtain the symbols, and the address and size of the code and data segments. The class symbol (defined in `util/h/symbol.h`) provides a representation for symbols. The file `util/h/Object.h` defines abstract classes for object files. Each platform has its own implementation: `Object-elf32.h` (for Solaris 2.x), and `Object-xcoff.h` (for AIX).

Once the symbol table is processed, the functions of the application process are defined. The class `pdFunction` provides a representation for functions. For each function, the method `findInstPoints` of the class `pdFunction` is invoked to find the instrumentation points for that function. The method `findInstPoints()` has one implementation for each architecture supported by Paradyn (currently sparc, mips, x86, and power). The implementations are in files `inst-sparc.C`, `inst-mips.C`, `inst-x86.C`, and `inst-power.C`

Class `instPoint` provides a representation for instrumentation points, defining the address of the point, the instructions to be relocated, and other relevant information. The class is defined in the architecture dependent files (`inst-sparc.C`, `inst-power.C` and `inst-x86.C`).

5.4 Shared-object processing

On Solaris platforms, Paradyn supports instrumentation of dynamic executables. A *dynamic executable* is one that is created by dynamically linking shared libraries (called *shared objects*). When the Paradyn daemon processes an `a.out` file of a dynamic executable, many of the symbols are undefined. These undefined symbols are from shared objects that are bound at runtime by the run-time linker.

Figure 5 shows the data structures used by the Paradyn daemon to keep track of shared object information for each process. This figure shows three process objects, one for each process run-

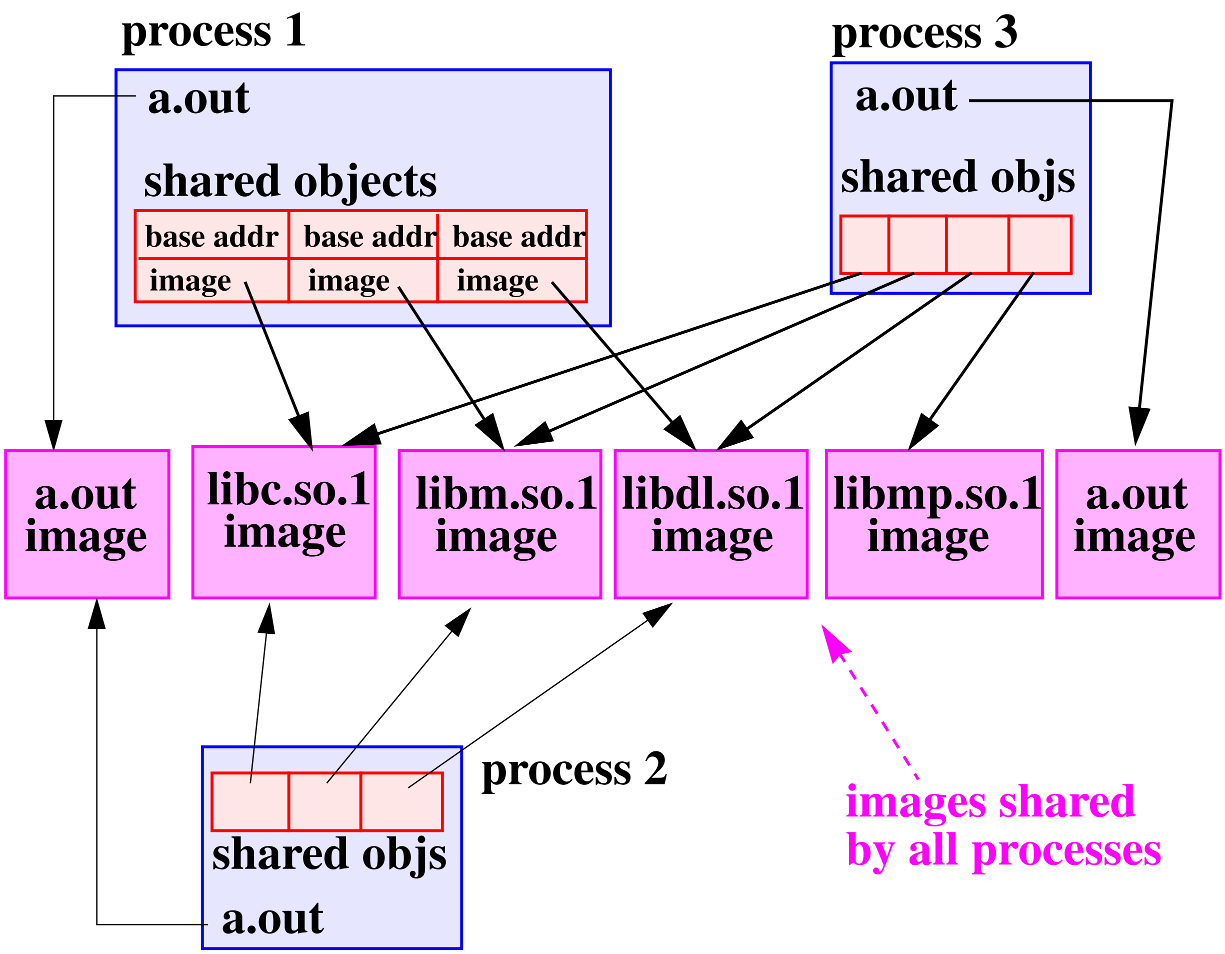


Figure 5: Process class and shared objects.
Process 1 and process 2 are the same executable and share `a.out` and shared object images. Process 3 is a different executable, running on the same host, which has some of the same shared object images as process 1 and 2, but a different `a.out` image.

ning on the host. Each process contains pointers to image structures. There is one image object for each unique executable file and shared object file processed by the Paradyn daemon. In this example, process 1 and process 2 are executing the same `a.out` files; they both contain pointers to the same `a.out` image. Process 3 is executing a different `a.out` file; it contains a pointer to a different `a.out` image. Each process object also contains a list of pointers to shared object images and a list of base addresses associated with these shared objects. Since two different executables can have the same shared object mapped into their address space at different addresses, the addresses of the instrumentation points of functions in shared objects may differ across processes. Rather than create multiple image objects for shared object files, each process keeps track of the base address of where it has the shared object mapped and then contains a pointer to the shared object's image. This way, only one image object needs to be created for each unique shared object or `a.out`.

Figure 6 shows the relationship between the `image`, `module`, `pdFunction`, and `instPoint` classes in the Paradyn daemon. Each image contains a set of modules, and each module consists of a set of functions. For each such function, a `pdFunction` object is created. This class contains information about each function, such as the function's name, address, and size. Each function also contains several instrumentation points. Currently, function exit, function entry, and pre- and post-call site instrumentation points are defined for each function. Paradyn creates an `instPoint` object for each of these instrumentation points.

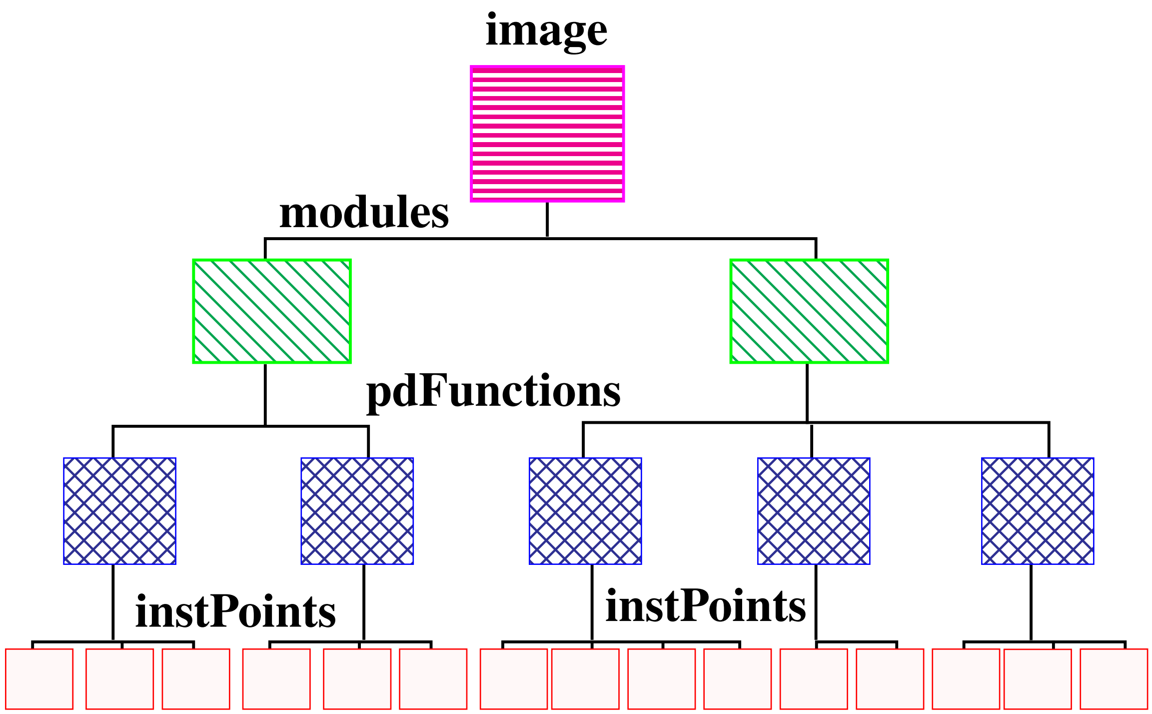


Figure 6: image, module, pdFunction, and instPoint classes.
Each image consists of a number of modules, each module consists of functions, and each function consists of a number of instrumentation points.

All address information stored in `instPoint` and `pdFunction` objects is kept relative to the image in which it is contained. This means that when inserting instrumentation into functions that are contained in a shared object, the base address value stored in the process object must be added to the address in the instPoint to find the correct location to write to in the process's address space. As a result, new `instPoint`, `pdFunction`, and `image` objects do not need to be created for every process that dynamically links a particular shared object.

Metric Evaluation and Code Generation

When a user or the Performance Consultant enable a metric/focus pair, the daemon must evaluate the metric, generate code, and insert instrumentation into the application process. Most of the code to do the metric evaluation is in file `mdl.C`. The metric is evaluated producing an intermediate code representation in the form of abstract syntax trees (class `AstNode` defined in file ast.h/ast.C). The abstract trees are then translated into machine code, which can be inserted into the application processes. There are different implementations of the code generator, one for each supported architecture, in files inst-sparc.C, inst-power.C and inst-x86.C.

Each metric/focus pair is associated with counters or timers, which are objects allocated in the inferior heap and operated by the instrumentation code inserted in the application process. Each allocated timer or counter is represented in the Paradyn daemon by an object of class timerHandle or intCounterHandle, defined in inst.h.

The class instInstance, defined in instP.h, provides a representation for instrumentation instances (a chunk of code inserted at some instrumentation point in a process). The functions addInstFunc() and deleteInstFunc(), defined in inst.C, are used to insert and delete instrumentation instances in an application process. addInstFunc() allocates base and mini-trampolines as needed, generates branches from the instrumentation points to the base trampolines, and from trampolines to other trampolines.

Each enabled metric/focus pair is represented by an object of class metricDefinitionNode (defined in metric.h). There are two types of metricDefinitionNodes, aggregates and non-aggregates. A metric/focus pair is always represented by an aggregate metricDefinitionNode. Each aggregate metricDefinitionNode has one or more components associated with application processes. The components are non-aggregate metricDefinitionNodes. A component metricDefinitionNode can belong to one or more aggregate metricDefinitionNode.

Each component is associated with a timer or counter, and for each component, instrumentation is inserted into the application process to periodically sample the value of the timer or counter, writing the values in a pipe that is read by the daemon. Each timer or counter has an unique identifier, that associates that timer or counter with a unique component. Once a sample is received, this identifier is used to find the corresponding component. The value is passed from the component to its parent metricDefinitionNode, aggregated with the values from other components, and forwarded to the Paradyn front-end.

Figure 7 shows the `metricDefinitionNode` data structure and its relation to other data structures. Note: the data structure hierarchy presented in Figure 7 only corresponds to that for the alarm sampling case (described in Section 5.5.2). These structures are currently evolving to cater for multi-threaded programs and the dyninst API.

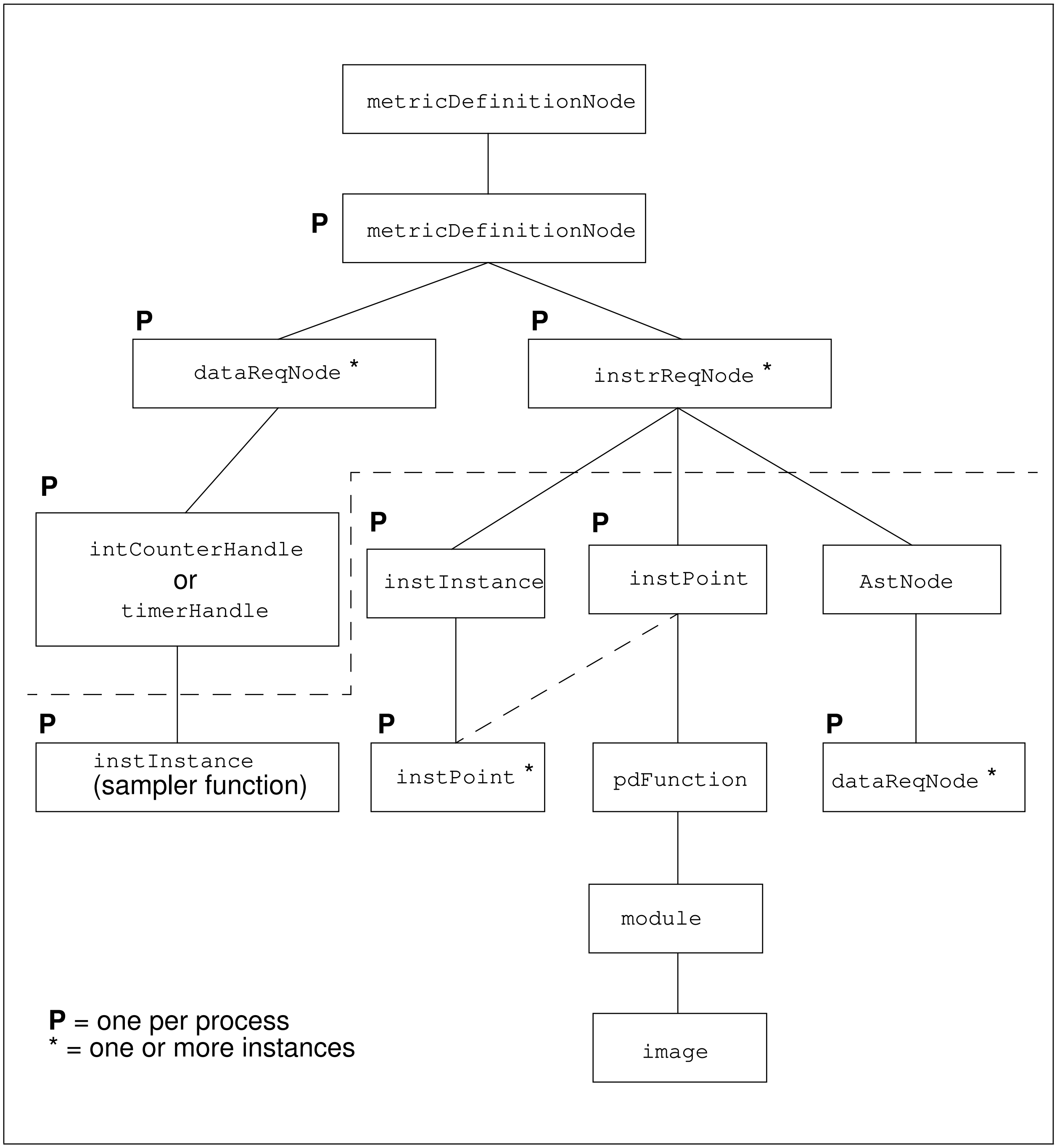


Figure 7: Data structures of the Paradyn daemon.

5.5 Performance data sampling

Performance data sampling is (along with dynamic instrumentation) one of the major tasks performed by the Paradyn daemon. Typically, instrumentation code inserted into an application will write performance data to various counters and timers. Periodically (up to 5 times per second), the Paradyn daemon is responsible for *sampling* these counters and timers, to be forwarded to the Paradyn front-end for processing by the Performance Consultant and visis.

Since the actual counters and timers reside in the application's address space, it is not immediately obvious how `paradynd` can *efficiently* sample them. Since a Paradyn daemon always runs on the same node as the application it is controlling, efficiency and perturbation are concerns. Certainly, `paradynd` could pause the process and extract the data using `ptrace` or `/proc`, but this would be too slow and intrusive. In this section, we will describe the two (very different) implementations of sampling currently implemented in `paradynd`. The first (and much more efficient) is called *shared-memory sampling*; the second is called *alarm-sampling*. Which one is used depends on the platform; for reasons discussed below, shared-memory sampling is implemented on most platforms.

5.5.1 Shared-memory sampling

In a shared-memory sampling `paradynd`, a shared-memory segment (created with `shmget()` on UNIX, `CreateFileMapping` under Windows NT) holds the counters and timers that need to be sampled. Both the application and `paradynd` in turn attach to this segment (using `shmat()` on UNIX, `OpenFileMapping` and `MapViewOfFile` on Windows NT). Since `paradynd` is attached to the segment, it can sample the counters and timers simply by reading directly from the segment's memory—the application need not know or care that it is being sampled. This contrasts with alarm sampling (Section 5.5.2), which requires the application to take an active role in sampling itself.

There are two complications that arise when implementing shared-memory sampling. First, since the application may be writing to a counter or timer while `paradynd` is sampling it, there needs to be some synchronization. Second, due to the semantics of sampling an active timer (one which has been started but not stopped), `paradynd` needs the ability to obtain the virtual (CPU) time of the application. Operating systems lacking such a primitive cannot use shared-memory sampling, and must use alarm sampling instead.

We now discuss these two complications in greater detail.

5.5.1.1 Synchronization issues for shared-memory sampling

Since instrumentation code inserted into an application may write to a counter or timer just as it is being sampled (read) by `paradynd`, care must be taken to ensure that a consistent value gets sampled.

For counters (integers), no special precautions are needed. If `paradynd` samples an integer while it is being modified, then either the old or new value will be sampled. Since both values are consistent, either is suitable.

Sampling timers is more complicated. Consider the pseudo-code for `startTimer`/`stopTimer` operations (Figure 8), and for `paradynd`'s shared-memory sampling of a timer (Figure 9).

```
    startTimer(tTimer *t) {
(1)     if (t->count++ == 0) {
(2)        t->start = get-current-time()
(3)     }
    }

    stopTimer(tTimer *t) {
(1)     if (--t->count == 0) {
(2)        t->total += get-current-time() - t->start
(3)     }

    }
```

Figure 8: Pseudo-code for startTimer and stopTimer operations

```
(1) sampled-value = t->total;
(2) if (t->count > 0) {
         // applic has done a startTimer but not (yet)
         // a corresponding stopTimer
(3)     sampled-value += get-remote-time() - t->start
     }
```

Figure 9: Pseudo-code for shared-memory sample of a timer

Assume that we are measuring the time spent in function `foo()`. To do this, the entry point of `foo()` is instrumented with `startTimer()` and the exit point is instrumented with `stopTimer()`. Furthermore, assume that `foo()` is a long-running function (say 5 minutes), so a long time can elapse between the `startTimer()` and `stopTimer()`. If sampling occurs after the timer was started, but before it was stopped, `t->total` will not include the time that has elapsed since the latest call `startTimer()`. Line 3 in Figure 9 ensures that the sampled value includes that interval. It assumes the existence of a *get-remote-time()* primitive—a way for `paradynd` to somehow obtain the current time of the application being measured. (The term *remote* comes from the fact that they're different processes.)

Now that we understand the basic code for `startTimer`, `stopTimer`, and sampling, we can explain the need for synchronization. Imagine if a sample is taken after the application has executed line 1 in Figure 8 but before it has executed any of line 2. In that case, `paradynd` will see the `count` field non-zero, so it will execute line 3 of Figure 9, using an undefined value of `t->start`!

Clearly, some kind of synchronization is needed. Note that an interrupt of some sort (such as a thread context switch or a signal handler) could happen at any time, and if such code re-enters the instrumentation code, deadlock would result. In short, using locks would render instrumentation code unsafe for reentrancy. Our solution involves *protector variables*; two counters which are part of the timer structure. The `startTimer` and `stopTimer` operations increment the first protector variable, then perform their work, then increment the second protector variable. The sampling routine reads the second protector variable, then the count, start, and total fields, and finally the first protector variable. Note that the protector variables are read in the reverse order that they are written. If the (sampled values of) the two protector variables are equal, then the sampled values of the count, start, and total fields are consistent. If not, the sample is thrown out, and the timer is re-sampled later. Figure 10 and Figure 11 show the new code for `startTimer`, `stopTimer`, and sampling.

```
startTimer(tTimer *t) {
   t->protector1++;
   if (t->count++ == 0) {
      t->start = get-current-time()
   }
   t->protector2++;
}

stopTimer(tTimer *t) {
   t->protector1++;
   if (--t->count == 0) {
      t->total += get-current-time() - t->start
   }
   t->protector2++;
}
```

Figure 10: Final pseudo-code for startTimer/stopTimer operations

```
(1) prot2 = t->protector2;
(2) sampled-value = t->total;
(3) if (t->count > 0)
(4)    sampled-value += get-remote-time() - t->start
(5) prot1 = t->protector1;
(6) if (prot1==prot2) {
(7)    use sampled-value; report it to front-end
(8) } else {
(9)    throw out the sample; re-sample later
    }
```

Figure 11: Final pseudo-code for timer sampling

5.5.1.2 The need for a *get-remote-time()* primitive

We have not found a way to implement the *get-remote-time()* primitive in line 3 of Figure 9 (and line 4 of Figure 11) on all platforms; this prevents shared-memory sampling from being ubiquitous. Simply put, there isn't a standard way in UNIX to obtain the virtual (CPU) time of another process (in this case, `paradynd` needs to obtain the virtual time of the application process). The `/proc` file system does provide a way; hence, shared-memory sampling is implemented on Solaris (both sparc and x86). Under Windows NT, we use the `GetProcessTimes` function to obtain the CPU time of another process.

5.5.1.3 Source code for shared-memory sampling

All code for shared-memory sampling resides in the `paradynd` directory; none is in the `dyninstAPI` directory. `fastInferiorHeapMgr.[hC]` contains a class for managing a shared-memory segment (with no interpretation of its contents). `superTable.[hC]`, `baseTable.[hC]`, `SuperVector.[hC]`, and `fastInferiorHeap.[hC]` manage allocation and deallocation of counters and timers from the shared-memory segment. Currently, we use more source code files than are needed; the extra levels of indirection — `superTable` contains several `baseTable`s, which in turn contains several superVectors — are put into place now in anticipation of later support of allocating timers and counters for multi-threaded programs.

The method doMajorSample() within fastInferiorHeap.C is a good place to start for understanding shared-memory sampling. It samples as many counters and timers as it can; those which it cannot sample (due to disagreeing "protector" variables—see Figure 11) are re-sampled later in doMinorSample(). The doMajorSample() routine is invoked by perfStream.C's checkAndDoShmSampling(), which is called each time through paradynd's main loop.

5.5.2 Alarm sampling

Since shared-memory sampling isn't ubiquitous, we have retained the method of sampling used in earlier releases of Paradyn; we call it *alarm-sampling* because sampling is triggered via a SIGALRM in the application. During initialization of the runtime library (Section 8), the application is set up so that it executes the routine DYNINSTalarmExpire() (in RTinst.c of the run-time instrumentation library, rtinst, directory) several times per second. This in turn calls DYNINSTreportSamples, which calls DYNINSTsampleValues(). DYNINSTsampleValues() is an interesting function; at first glance, it appears empty. However, in alarm sampling, paradynd actually instruments this routine to call DYNINSTreportCounter() or DYNINSTreportTimer() as appropriate for each counter and timer that needs to be sampled. These routines in turn call DYNINSTgenerateTraceRecord() to (rather inefficiently) send this information to the Paradyn daemon via a pipe. From there, alarm sampling is similar to shared-memory sampling — both portions of paradynd forward the sampled data to Paradyn by calling the updateValue() method of the metric instance (metric.C), which eventually forwards bulk data to Paradyn via the batchSampleDataCallbackFunc() Igen routine.

5.6 Retroactive instrumentation

[Relevant files: paradynd/src/metric.C contains most of the logic for this mechanism and dyninstAPI/src/inst-{platform}.C files contain helper functions.]

Retroactive (catchup or *ketchup* [sic]) instrumentation is a special mechanism to deal with a problem which arises with dynamic instrumentation. When a function is instrumented with code near the beginning of a function, and this instrumentation is inserted while the program is running, the possibility arises of the instrumentation being missed by the currently running function. In instrumentation where code inserted near the end relies on code inserted near the beginning, or where the function only runs once, the inserted code may enter an inconsistent state, or not ever be executed.

A good example is the timing of a function. At the beginning of the function, a timer is started. At the end, a timer is stopped. If the function is executing, the timer will not be started for this execution of the function: this is a problem if a single execution of the function runs for long periods of time.

The solution is to retroactively execute the snippets of instrumentation which have been missed at the time which the function is being instrumented. Unfortunately, it is impossible to know all of the prior execution history of an uninstrumented process. It is possible, however, to recreate a partial (minimal) history which corresponds to the functions currently found on the call-stack: to arrive at the current call-stack state, each function must have been entered (but not exited) and made a call to its successor (but not returned from such a call). Any corresponding entry-point and pre-call instrumentation snippets would have also been executed in a previously instrumented execution, and therefore these snippets should be retroactively executed.

Note that it is entirely likely that additional functions have been called and already exited, e.g., a call located earlier in a function than the call currently found on the stack, but there is insufficient residual evidence to reliably suggest that their associated instrumentation snippets should now be executed. (A complete control-flow graph for each function would allow some such cases to be determined, but many cases would require missing dynamic control-flow information.) This means that the retroactively-constructed instrumentation state is necessarily incomplete where there is an instrumentation dependence other than the following cases:

- (parent) function entry precedes any internal calls to (child) functions precedes function exit, and
- function pre-call precedes function post-call (within the context of any function).

Fortunately, these are exactly the relations typically used in instrumentation to delimit inclusive (i.e., entry to exit, or equivalently pre-call to post-call) and exclusive (i.e., entry to exit excluding internal calls) metrics for functions (or function calls). Other relations may be defined, such as between two calls or arbitrary points, however, there is insufficient residual information for retroactive instrumentation to be reliably used in such cases and conforming metrics must therefore not rely on associated snippets being executed.

Paradyn already contains a mechanism for causing code to be run in the inferior process: the inferior RPC. By using this mechanism when appropriate, we can preempt the execution of the current function and execute required snippets of retroactive instrumentation.

The following algorithm is used to determine if it is appropriate to launch a catchup inferior-RPC for a specific snippet of instrumentation:

- If the instrumentation is to be placed at the function entry point, and that function is currently anywhere on the call stack, a catchup inferiorRPC should be launched to execute it.
- If the instrumentation is to be placed just before a call site, and that call site is in fact on the stack, a catchup inferiorRPC should be launched for it. In other words, if the PC of the call site is on the stack, but not at the top.

This check is performed in the `process.C` file in the function `triggeredInStackFrame`.

There are a few additional aspects which need to be addressed:

- On some architectures (SPARC in particular), instrumentation must be deferred if it is not safe to insert code immediately. If this happens for instrumentation which depends on the instrumentation being considered for catchup, we must not do the catchup. Executing the early code without the late code may cause more inconsistencies than executing only the late code. A special case of this, is when the function at the top of the stack cannot be instrumented due to the PC currently being located within a potential instrumentation footprint: not only should catchup instrumentation not be executed for this particular function instance, it should also not be executed for any other instances of this function found lower on the stack, as the function itself is currently uninstrumentable, and there is correspondingly no catch-up to be done.
- On x86 architectures, where traps are used in tight instrumentation points (see Section 6), inferiorRPC execution may be interrupted by the delivery of a signal raised by a current trap instruction: since processing traps is relatively time-consuming, interruption at such points is quite likely. The usual trap handling by `DYNINSTtrapHandler`, which expects to be delivered

an instrumentation-trap PC value, must recognize and ignore an inferiorRPC-adjusted PC value, and resume execution of the inferiorRPC(s) before returning to re-execute (and handle) the interrupted trap (and its associated instrumentation). Note that traps in code executed by calls to functions from the inferiorRPC require appropriate handling.

- The ordering of catchup instrumentation within each function and on the stack can be very important. For each set of instrumentation snippets to be inserted, a list of instrumentation to be executed via inferiorRPC must be kept: ordering should be chronological with respect to the implied program execution (derived from the call stack) to arrive at the current state, i.e., starting from the base of the stack, each subsequent frame is considered in turn to decide whether to launch catchup inferiorRPCs for that frame. When the catchup inferiorRPCs are launched, they must follow this order.
- The address of the PC at each stack frame must be mappable to the function in the program to which it corresponds. In the cases where the PC is instead within our instrumentation code, we must properly find the function to which that instrumentation corresponds.
- When we are within instrumentation code on the stack, we must amend the above check for being within a call site to take into account that we usually relocate the `call` instruction itself to within the instrumentation code base-trampoline.
- Furthermore, if the pending instrumentation snippet happens to be new/additional instrumentation for the current instrumentation point, careful analysis needs to be made to determine whether catchup execution is required. If it has been added to the existing base/mini-trampoline infrastructure at a point *after* the current location, then it will be executed normally and catchup execution is inappropiate, otherwise a catchup inferiorRPC should be launched to execute it. For example, if the call-stack contains an instrumentation mini-trampoline for the same instrumentation point as the pending instrumentation snippet, then catchup execution is required for *prepended* snippets but should not be executed for *appended* snippets.
- The catchup inferiorRPCs must be executed immediately, while the inferior is paused. If the inferior is allowed to execute anything other than our inferior RPC, the function may exit and re-enter before the inferior RPC is launched. If this happens, it is quite possible for the inferiorRPC to run while the same instrumentation is executing within the inferior on the same data. This is particularly bad with our timers, which have critical sections which assert on failure. Therefore, we cannot finish the checking of instrumentation and rely on the main `paradynd` loop to launch the inferiorRPCs, we must make a special loop which launches them, and which keeps the inferior process paused between inferiorRPCs. Note that it is fine for a catchup inferiorRPC to be launched to start or stop a timer when interrupting a timer operation corresponding to a distinct metric/focus instance, but not the same metric/focus instance since in that case the timers are identical. Luckily, if we find ourself interrupting one of our timer operations, then we can be assured that we are executing an already-instrumented function and there is consequently no need for retroactive instrumentation.
- In the case of successful instrumentation of an on-stack function with an exclusive metric (which relocates it's currently active call instruction to our instrumentation base-trampoline) and execution of appropriate (entry and pre-call) retroactive instrumentation for it, it is ultimately necessary to update the return address of the succeeding stack frame to return to immediately after the call instruction relocated in the base-trampoline instead of the now-overwritten location. This thereby ensures the usual execution of corresponding post-call

instrumentation, which would otherwise be missed, or worse, the resumption of execution within a now-corrupted instruction sequence.

An example of retroactive function instrumentation is shown in Figure 12

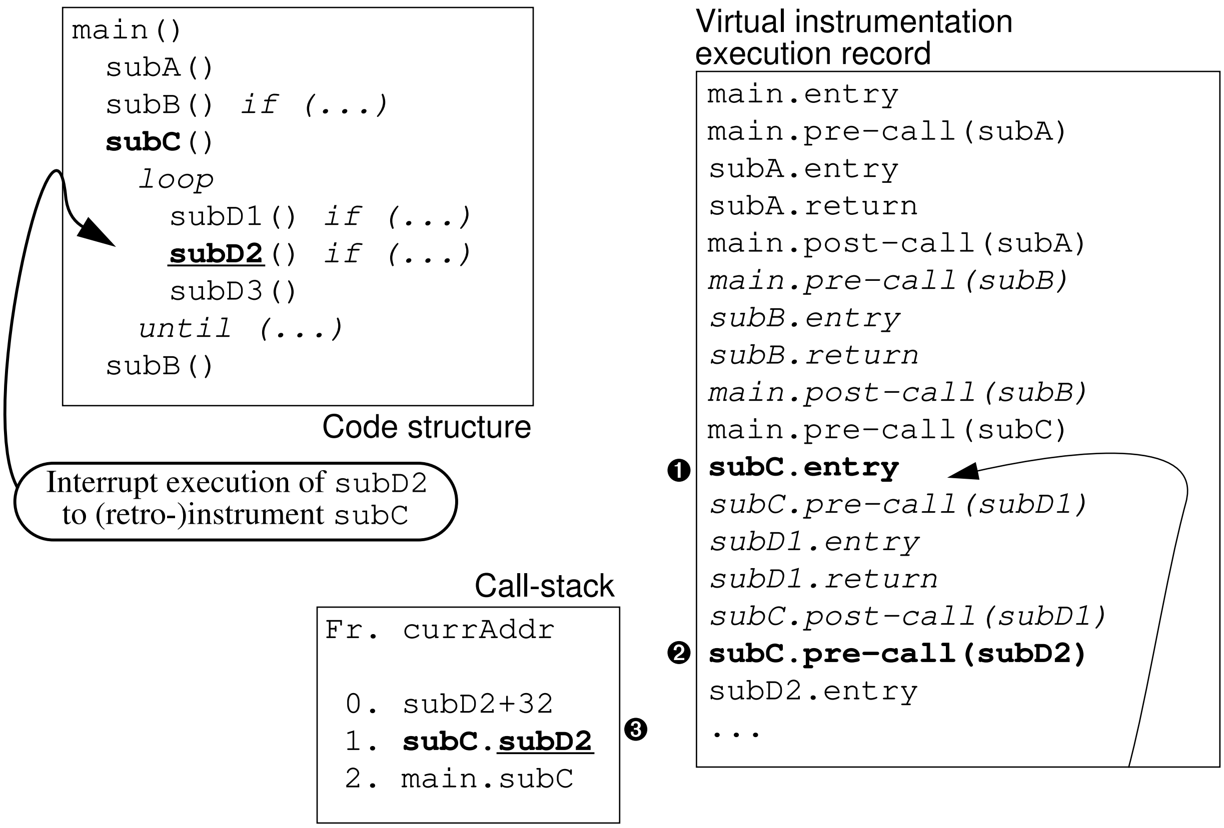


Figure 12: Retroactive instrumentation example.

The program has been interrupted during the execution of `subD2` ***(with the call-stack as shown) with a request to instrument*** `subC`***. In addition to instrumenting the appropriate points in*** `subC`***, to support the illusion that*** `subC` ***was already instrumented it is necessary to retroactively execute its entry-point ❶ and*** `subD2`***-precall ❷ instrumentation snippets (if they constitute part of the instrumentation request). Italicized parts of the virtual execution record can't be recovered from available state information. To ensure that following the completion of*** `subD2`***, execution will correctly continue with any*** `subD2`***-postcall instrumentation snippets, it is also necessary to update the return address of*** `subD2`***'s stackframe ❸ with that of the base-trampoline now containing the relocated call instruction.***

5.7 Dynamic Heaps

Paradyn and Dyninst use a dynamic heap to store code and data for instrumentation and inferior RPCs in the application process. Dynamic heaps enable an arbitrary amount of instrumentation code to be placed in the application process. Also, on platforms with restricted single-instruction branch ranges (*e.g.*, RISC processors), they can be directed to allocate memory near a particular address in memory. These directed allocations are used to place base trampolines within the range of a single instruction branch from the corresponding instrumentation point. Dynamic heaps are currently not used on NT.

All memory allocation requests in the Paradyn daemon and Dyninst mutator are made through `inferiorMalloc`, defined in `process.C`. `inferiorMalloc` maintains the inferiorHeap data structure, which organizes the memory that has been allocated in the application process for Paradyn/Dyninst, and includes a list of free memory. `inferiorMalloc` takes a parameter named `size`, the number of bytes to be allocated in the application process. For directed allocation it also takes an optional `near` parameter, a pointer to which the requested memory should be close. (The definition of close is platform-specific and defined at compile-time, as explained below.) Finally, `inferiorMalloc` takes a type parameter to specify the type of heap segment used to satisfy the request. The type parameter includes all heap segments except the low memory heap, which is explained below.

Ordinarily, `inferiorMalloc` can satisfy a request by finding suitable space in the inferiorHeap free list. When it cannot, it makes an inferior RPC to `DYNINSTos_malloc` in the run-time instrumentation library (defined in `RTheap.c`). `DYNINSTos_malloc` allocates new segments of the address space of the application process for use as heap segments by Paradyn/Dyninst. `inferiorMalloc` calls `DYNINSTos_malloc` in two circumstances: (1) when there is not enough memory in the free list to satisfy the request, or (2) when the request is directed, but there is not enough free memory within the range of the near pointer. `DYNINSTos_malloc` takes three parameters: the number of bytes to allocate, and low and high address boundaries. `DYNINSTos_malloc` is not intended to satisfy a single `inferiorMalloc` request, but rather to allocate new heap segments in the inferior process from which subsequent allocation requests can be satisfied; the size parameter is thus usually much greater than that of the current `inferiorMalloc` request. On directed allocations, the address boundaries are the range of the address space in which the new memory can be allocated. When the request is undirected, they are opened to the entire address space. When an inferior RPC to `DYNINSTos_malloc` returns, `inferiorMalloc` satisfies its current request from the new heap segment, and adds the remainder of the new segment to the free list.

`inferiorMalloc` is aggressive in its use of `DYNINSTos_malloc`. If `DYNINSTos_malloc` cannot satisfy its first request, `inferiorMalloc` makes several additional calls to `DYNINSTos_malloc` with increasingly relaxed parameters. For example, it will reduce the request size of the new segment, and lift the address space boundary restriction. This retry sequence happens in the `for` loop of `inferiorMalloc`.

`DYNINSTos_malloc` has two mechanisms for allocating new memory. On platforms that use directed allocation, it calls `malloc` if the near pointer is within the range of the heap of the inferior process. Otherwise, `DYNINSTos_malloc` calls `constrained_mmap`, which in turn calls `mmap`. `constrained_mmap` reads `/proc` to determine the layout of its own address space. The platform-specific memory information returned from `/proc` is translated into an array of a generic structure called `dyninstmm_t`. Within this array `constrained_mmap` searches for a hole into which memory of size and location satisfying the `DYNINSTos_malloc` request can be allocated. It then calls `mmap` to try to allocate that memory. As some platforms may restrict the location of mmapped memory, `constrained_mmap` makes a call to mmap for every hole it finds until one is successful. If none are successful it returns an error to `inferiorMalloc` (which may retry `DYNINSTos_malloc` using relaxed parameters). When it is successful, `DYNINSTos_malloc` returns the address of the first byte in the newly allocated memory.

On all platform the run-time instrumentation library contains a static buffer of heap space called DYNINSTdata. It is added to the free list used by inferiorMalloc, and will be used to satisfy memory allocation requests if it is sufficiently large and, for directed allocations, suitably located. On platforms that don't use directed allocations all inferiorMalloc requests are satisfied in the static heap before a dynamic heap is allocated (*i.e.*, before a call to DYNINSTos_malloc is made).

From DYNINSTdata a small static buffer called the low memory buffer is reserved during heap initialization. Its purpose is to ensure there is always enough space in the inferior process to make a new dynamic heap allocation. The only time the low memory buffer is used is when an inferior RPC to DYNINSTos_malloc is made. It is distinguished from the other heap segments by its heap type lowmemHeap, which no other heap segment has.

Restrictions in the range of address space that can be used to allocate a new heap segments are determined in two ways. First, the caller of inferiorMalloc can use directed allocation to make an explict restriction. The range of a directed allocation depends on the range of control transfer instructions of the processor. This range is computed by the region_lo and region_hi macros in the architecture header files (*e.g.*, arch-sparc.h). If the control transfer range is unrestricted, these macros are defined to include the entire user-accessible portion of the address space. Second, platform-specific characteristics of the address space may preclude some ranges from being safe places for new heap segments. For example, it is not safe to allocate new segments near the top of the stack, as the stack may eventually grow into the segment. The run-time instrumentation library set these limits with two variables, DYNINSTheap_loAddr and DYNINSTheap_hiAddr. Their values are checked by DYNINSTos_malloc and they take precedence over directed allocation constraints. On some platforms (*e.g.*, Solaris) these values can vary from process to process, and are thus initialized at run time after consultation with /proc.

5.8 Trampoline Guards

The basic trampoline structure has one dangerous flaw: it is possible to inadvertently cause an infinite recursion in the instrumentation which will cause the instrumented program to crash. Specifically, instrumentation can never safely call any other function (even in a library) which is instrumented. Making instrumentation safe in this manner is both difficult to ensure and limiting.

To avoid this effect, Paradyn now includes guards in the trampoline structure which will prevent any recursion from taking place. These guards detect if the current base tramp is being executed inside instrumentation, and if so skips the instrumentation contained within the trampoline. The end result of this is instrumentation can call any function with impunity, without having to worry about side effects. Currently these guards are implemented on AIX, IRIX, and SPARC-Solaris. Support on other platforms will be available shortly.

To motivate our use of trampoline guards, let's use an example. A typical metric used by Paradyn is the io-wait metric, which instruments the system call write() with a wall timer. Inside the instrumentation, we use fprintf() to report timer rollbacks and we assert that the timer is not started twice without being stopped. With this setup, any timer rollback would cause an infinite recursion in the process being instrumented. Specifically, the call to fprintf() would cause the timer to be started again when write() was called. This would trigger the assertion, which would print an error and terminate the program. Unfortunately, the act of printing the error would cause the timer to be started yet again and trigger another assert. Using the trampoline guards in this case will ensure that the call to `fprintf()` within the timer routine will not set off any other instrumentation.

On platforms where the guards are implemented, an additional word of memory is allocated in the processes address space. This flag (`*process::trampFlagGuardAddr`) is used to store whether the current execution point is inside a base tramp or not. The value actually used is platform-dependent. When a base tramp is entered, the value of the flag is checked. If the flag is true, then the intrumentation is skipped. Otherwise, the flag is set to true and the instrumentation entered.

The trampoline guards have the following general structure:

```
<save registers>
if (flag == true) then skip to <restore registers>
<set flag to true>
<enter minitramp>
<set flag to false>
<restore registers>
```

Note that the guard code is added at the base trampoline level, so it is correct to speak of the guard code at an instrumentation point, rather than the guard code for a piece of instrumentation code. All the mini-tramps that are called by the same base tramp are guarded by an unique piece of guard code that resides in the base tramp. At the present time, there is no way to guard (against recursion) only certain minitramps. Whether a base tramp is guarded or not is determined when it is first inserted and is unchangeable.

By default, all Paradyn instrumentation is inserted with the trampoline guard enabled. This also has another strong benefit: Paradyn can now call functions without disturbing the data being reported about the inferior process. For example, if a piece of instrumentation calls an instrumented function which is particularly CPU hungry, then the CPU usage for the function when called by instrumentation will not be reported.

5.9 Timer Levels

Paradyn includes support for two timer levels for both process and wall timers. This allows unique time querying functions, native time units, native time bases, availability test functions, and other features to be associated with timer levels. One of the levels is the hardware timer level which can be used for time querying functions that are less in time cost or greater in granularity than the software timer level, typical when directly accessing the hardware. The other timer level is the software timer level, which is for time querying functions that access the time through software and is less desirable than a possible hardware timer level. The notions of software and hardware in regards to timer levels were not meant to be rigid, but serve to inform that the hardware timer level has a smaller time cost and/or higher granularity than the software timer level. There are platforms which don't have both timer levels implemented, yet at least one timer level needs to be implemented for each platform for process and wall timers. An example of a platform with only one timer level is the sparc-solaris version of Paradyn. The software timer level for this platform has low time cost and high granularity so hardware timer levels for process and wall timers are not implemented on this platform. At any point in time, there might also exist platforms for which a hardware level version of a timer is not possible with the platform's current state of technology.

A boolean availability test function is associated with each timer level in order to aid in choosing which timer level to use. The hardware timer level will be chosen if the availability test function informs that the level is available. If the level is not available, the software timer level will be chosen (assuming it is available). However, it is possible to override this mechanism by setting the environment variables PD_SOFTWARE_LEVEL_WALL_TIMER or PD_SOFTWARE_LEVEL_CPU_TIMER which will cause the software level timer to be chosen over the hardware level timer. The timer level that is chosen can be displayed in the terminal by setting the environment variable PD_SHOW_TIMER_INFO.

One benefit that came with this multiple timer level feature, though not inherently related, is that the rtinst library no longer needs to convert time into a standard time unit (used to be microseconds). Now time querying functions in the rtinst library can return time in the native time unit that was queried. The daemon now will do the appropriate conversion from the native time unit into a generic time object (timeStamp for wall time and timeLength for cpu time). This offloads work from the time querying functions in the rtinst library and hence the application also. Sampling by the daemon occurs no more frequently than 5 times per second which is much less frequent than the number of times an application calls a time querying function when instrumentation is being done.

The notion of a level in Paradyn is represented by a class called timeMechanism (paradynd/src/timeMechanism.h). The notion of a set of levels is represented by a class called timeManager (paradynd/src/timeMgr.h). The timeMgr handles all interaction to a timer level and therefore there should not be a need to access a timeMechanism object directly. For example, the timeMgr class has member functions for installing a timer level (installLevel), determining the best available timer level (determineBestLevels), or retrieving the native or converted time (getTime, getRawTime). The timeMgr class was made a template class in order to handle different requirements for interacting with timer levels. The first template argument is used for the different contexts for which given function pointers may be called. For example, the function pointer for a process time querying function is a member of the process class. This function pointer needs to be called differently than the function pointer for the wall time querying function, which is not a member function. The second template argument specifies the type of the argument required when calling the time querying function. For example, the process time querying function for the multi-threaded Solaris version of Paradyn requires the light weight process id to be passed as an argument.

In an execution of Paradyn, there will always be one and only one instantiation of a timeMgr for handling the wall timers, named wallTimeMgr. Functions for instantiating and accessing the wallTimeMgr are in paradynd/src/init.[hC]. There, will be one instantiation of a timeMgr for every process a daemon is monitoring. The timeMgr instantiation for process timers is the variable cpuTimeMgr which is a member of the process class. The process class has functions for interacting with the cpuTimeMgr such as initCpuTimeMgr, getCpuTime, and getRawCpuTime.

The selection of which timer level to choose is done solely by the daemon. The daemon than informs the rtinst library of which time querying function to use by assigning the function pointer pDYNINSTgetCPUtime or pDYNINSTgetWalltime in rtinst the address of the chosen time querying function. The daemon verifies that the chosen timer level is also available in the rtinst library by checking the value of the rtinst variables hintBestCpuTimerLevel and hintBestWallTimerLevel.

Implementing a new timer level

In order to implement a new timer level, there are particular functions that need to be modified or added. For both process and wall timers, in the RTinst library, in the appropriate RTetc-<platform>.c file, the time querying function associated with the level (DYNINSTgetCPUtime_hw or DYNINSTgetCPUtime_sw) will need to be implemented. In this same file, in the PARADYNos_init function, the appropriate variable hintBestWallTimerLevel or hintBestCpuTimerLevel needs to be assigned the macro define HARDWARE_TIMER_LEVEL or SOFTWARE_TIMER_LEVEL depending on whether the hardware timer level is available.

For implementing a process timer there are changes that need to be made in the daemon also. In the operating system specific file in dyninstAPI/src (eg. linux.C) the function process::initCpuTimeMgrPlt needs to be updated so that the new timer level is installed in the cpuTimerMgr. Also in this file, the function process::getRawCpuTime_hw or process::getRawCpuTime_sw that corresponds to the timer level, needs to be implemented. These functions are the process time querying functions used by the daemon. Also, support functions and class variables may need to be added to the process class. This may involve an update to process.h for adding member functions or variables to the process class.

For implementing a wall timer, in the directory paradynd/src in the init-<platform>.C file, in the function initWallTimeMgrPlt, a level needs to be installed for the new timer level into the wallTimeMgr. This file is also where wall time querying and availability test functions should be implemented. For example, for the NT platform in init-winnt.C, the function dm_isTSCAvail tests whether the hardware level wall timer is available and the function dm_getTSC queries the hardware level wall time for the daemon.

6 X86 PORT

Instruction representation

The representation of x86 instructions is different from other platforms. Because the size of instructions are variable, we represent an instruction by an object of class instruction, which is defined in the file `arch-x86.h`. The representation includes a type descriptor, the size of the instruction in bytes, and a pointer to the actual instruction (in the memory mapped executable image).

When instructions are processed, we need to decode instructions in order to find the size and type information about each instruction. The instruction decoder is implemented in the file `arch-x86.C`. The decoder is invoked through a method in class instruction (`getNextInstruction`).

Parsing the executable image

As in other platforms, the executable is parsed one function at a time. We start at the beginning of each function and decode instructions sequentially until we reach the end of the function (which is defined by the address of the next symbol in the symbol table).

The entry point is defined as the first instruction in the function. Call points are call instructions. Return points are return instructions and jumps that leave the current function. There is no check for tail-call optimization on the x86.

Data mixed with code (e.g. jump tables) is a problem as they could cause us to decode instructions incorrectly. We use some heuristics to try to identify some jump tables that may be within the code. We look for indirect jump instructions of the form

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

where `reg` is one of the general registers and *addr* an immediate address, which is the base the jump table. If the base address is within the current function and precedes the jump instruction, we may have parsed instructions incorrectly and we don't instrument the current function. In most cases, the jump table is just after the jump instruction, or near the end of the function. In this cases we can try to guess the size of the table by looking at the words following the base address and checking if their contents is an address within the current function. If so, we assume that it is part of the jump table and keep looking at the following addresses until we find an address that is not within the current function. Those locations that are found to be part of the symbol table are skipped. While this heuristic can not guarantee that we can find all jump tables, it is effective in detecting the jump tables generated by many compilers. A more general solution to this problem would require data and control flow analysis of the executable.

Since instrumentation points may not have enough bytes to replace with a jump (5 bytes), we may need to get additional instructions and add them to the smaller points. We can get instructions from before or after the point. For the entry point, we can only get extra instructions from after the point. For return we would usually only get instructions before the return, but since it is common to have nops or `int3` instructions after a return, we can also use those instructions. For call sites, we only get instructions from before the point for reasons that are explained later (although most calls are 5 bytes and don't need extra instructions).

We must check that there are no jumps into the middle of a sequence of instructions that we add to a point. To do that, we keep a list of all known jump targets, and check the instruction sequences against this list. The target of all direct jumps found while the image is parsed are added to the list, and also the addresses in the jump tables found by the heuristic described above. Since we can have jumps to other functions, we add the necessary number of instructions to the point here, and check later, when the point is instrumented that there are no jumps to the middle of the instruction sequence. Since there may be some indirect jumps for which we don't know the target, we may have problems if we use an instruction sequence that can be the target of an indirect jump. With the jump table heuristic above, we should be able to handle most cases.

Inserting instrumentation

Whenever we need to replace a sequence of multiple instructions, we must check that there are no jumps into the middle of the instructions. To do that, we keep a list of all known jump targets, and check the instruction sequences against this list. The list contains the target of all direct jumps found while the image is parsed, and the addresses found by the heuristic to skip jump tables (described above). Since there may be some indirect jumps for which we don't know the target, we may have problems if we use an instruction sequence that can be the target of an indirect jump. With the jump table heuristic, we should be able to handle most cases.

When we replace an instruction sequence with a jump, we must also check that the program is not currently executing in the middle of the sequence. Since we are modifying that sequence, we could execute the wrong code. If this is the case, we change the program counter to the address of the relocated instruction in the basetramp. We could also have a problem if we had calls in the middle of an instruction sequence. The call could be active, and eventually the callee could return to an invalid location. For this reason, we avoid putting calls in the middle of instructions sequences that are replaced with jumps. We should also check all possible contexts of the application (threads and exceptions), but this is not being done yet.

In some cases, we can't find enough instructions to replace with a jump, but we may be able to insert an indirection. We take enough space for two jumps in the entry point (if possible). If another point does not have enough space for a long jump (5 bytes), but has enough space for a short jump (2 bytes), and that point is within a short distance from the entry point (less than 128 bytes), we can insert a jump to the basetramp in the second jump slot of the entry point, and insert a short jump to this slot. In this case, whenever we activate the second point (which uses the entry point slot), we must also activate the entry point, even if there are no instrumentation requests for the entry point

Function Relocation

If there are not enough instructions to replace with a jump, and we cannot make an indirect jump to the basetramp, we expand and relocate the function. This involves creating a copy of the function with nop instructions inserted into those instrumentation points that are too small to replace with a jump instruction. The nops expand the instrumentation points, making them large enough to hold 5-byte jump instructions. A jump to the expanded copy of the function is then placed at the entry to the original function. It should be noted that function expansion and relocation often causes the targets of PC-relative call and jump instructions to be incorrect, since the relative locations of these instructions has changed. This requires that we update the displacements of some PC-relative instructions. In the extreme case, where the target address of a 2-byte jump instruction is no longer within the range of the jump, we must change the 2-byte jump into a 5-byte jump.

The relocation of a function is done the first time a request to instrument the function is made. This occurs even if the current instrumentation request is for an instrumentation point that is large enough to replace with a jump. Currently there are three types of functions that we do not relocate. These are functions that contain a jump table, are too small (less than 5 bytes), or are too large (greater than 16384 bytes). In such functions, when we can't find enough instructions to replace with a jump, we must insert a breakpoint instruction (`int3`). When the breakpoint is executed it generates an exception that can be caught in the application or by the Paradyn daemon. The address of the base tramp is entered into a hash-table, that is used by the breakpoint handler to find the address of the base tramp. The handler then changes the context of the application so that it executes the base tramp. On Solaris, the handler runs in the application, while in Windows NT and Linux, it runs in `paradynd`.

Base trampoline

The base trampoline for the x86 has some differences from other platforms. First the relocated instructions do not always go in the same place. Only the instruction at the point goes at the usual slot for relocated instructions, in the middle of the base tramp code. Any extra instructions from before the point, are relocated to the beginning of the trampoline, and extra instructions from after the point are relocated to the end of the trampoline, right before the jump back to the application code. One of the advantages of placing the instructions in different points of the base trampoline is that we can add jump instructions to a point when we need extra instructions. For example, if we have a return after a conditional jump, we can use that jump to insert a jump to the base trampoline for the return. Since the jump is relocated to the beginning of the trampoline, if the jump is taken the rest of the trampoline code will not be executed (which is the right thing).

The base trampoline for the x86 is not of fixed size, like in other platforms, since the size of the relocated instructions is variable. Unlike in the other platforms, where there is a template for the base trampoline code, in the x86 the code is generated when the trampoline is created.

There is one special case when the instruction at the point is a conditional jump. We relocate it to the top of the base trampoline, and change the code so that the trampoline is executed only if the branch is taken.

Code generation

The code generated for the x86 platform uses virtual registers, that are allocated on the stack. They are addressed as an offset from the frame pointer register (EBP). The virtual register are allocated on the base trampoline.

Example

Here we show the instrumentation of a function, and sample trampoline code.

```
f:    pushl %ebp
f+1:  movl %esp,%ebp
f+3:  subl $0x4,%esp
f+6:  movl $0x0,0xfffffffc(%ebp)
f+13: subl %eax,%eax
f+15: incl %eax
f+16: movl %eax,0xfffffffc(%ebp)
f+19: cmpl $0x3e8,%eax
f+24: jl <f+15>
f+26: subl %eax,%eax
f+28: leave
f+29: ret
f+30: nop
f+31: nop
```

This function has two instrumentation points: the entry point (the first instruction, at address f) and the return point (the `ret` instruction, at address f+29). Both places have one byte instructions, that can't be replaced by a jump. For the entry point, we can add the instructions at f+1 and f+3, which sum to a total of 6 bytes. For the return point, we need to add instructions from before the return. We need to add the instructions at f+28 (`leave`), f+26 (`subl`), and f+24 (`jl`).

After the insertion of instrumentation for the entry and return points, the function will look like:

```
f:    jmp baseTramp0
f+5:  *** garbage ***
f+6:  movl $0x0,0xfffffffc(%ebp)
f+13: subl %eax,%eax
f+15: incl %eax
f+16: movl %eax,0xfffffffc(%ebp)
f+19: cmpl $0x3e8,%eax
f+24: jmp baseTramp1
f+29: ret
f+30: nop
f+31: nop
```

(Note that most debuggers will not disassemble this code correctly, they get confused by the garbage at location f+5).

Base trampoline for the entry point:

```
// relocated extra instructions from before the point go here
// there are no extra instructions from before the point in this case

// pre-point instrumentation
baseTramp0:       jmp <baseTramp0+5>// slot to skip pre instrumentation
baseTramp0+5:     pushl %ebp        // set-up stack frame for minitramps
baseTramp0+6:     movl %esp,%ebp
baseTramp0+8:     subl $0x80, %esp  // allocate virtual registers
baseTramp0+14:    pusha             // save registers
baseTramp0+15:    pushf
baseTramp0+16:    jmp <minitramp>   // jump to minitramp
baseTramp0+21:    popf              // restore registers
baseTramp0+22:    popa
baseTramp0+23:    leave             // undo minitramp stack frame
baseTramp0+24:    addl 0x29, DYNINSTobsCost // update observed cost

// relocated instruction at entry point
baseTramp0+34:    pushl %ebp

// post-point instrumentation
baseTramp0+35:    jmp <baseTramp0+51> // skip post-instrumentation
baseTramp0+40:    pushl %ebp        // set-up stack frame for minitramps
baseTramp0+41:    movl %esp,%ebp
baseTramp0+43:    subl 0x80, %esp   // allocate virtual registers
baseTramp0+49:    pusha             // save registers
baseTramp0+50:    pushf
```

```
baseTramp0+51:    jmp <baseTramp0+48> // slot for jump to minitramp
baseTramp0+56:    popf              // restore registers
baseTramp0+57:    popa
baseTramp0+58:    leave             // undo minitramp stack frame

// relocated extra instructions at entry point
baseTramp0+59:    movl %esp,%ebp
baseTramp0+61:    subl $0x4,%esp

// jump back to application code
baseTramp0+64:    jmp 0x805038a <f+6>
```

The base tramp for the return point:

```
// relocate instructions before the point
baseTramp1+0:     jl 0x8050393 <f+15>
baseTramp1+6:     subl %eax,%eax
baseTramp1+8:     leave

// pre-point instrumentation
baseTramp1+9:     jmp <baseTramp0+5> // slot to skip pre instrumentation
baseTramp1+14:    pushl %ebp        // set-up stack frame for minitramps
baseTramp1+15:    movl %esp,%ebp
baseTramp1+17:    subl 0x80, %esp   // allocate virtual registers
baseTramp1+23:    pusha             // save registers
baseTramp1+24:    pushf
baseTramp1+25:    jmp <minitramp>   // jump to minitramp
baseTramp1+30:    popf              // restore registers
baseTramp1+31:    popa
baseTramp1+32:    leave             // undo minitramp stack frame
baseTramp1+33:    addl 0x29,DYNINSTobsCost // update observed cost
// relocated instruction at point
baseTramp1+43:    ret

// post instrumentation -- never reached in this case
baseTramp1+44:    jmp <baseTramp1+63>//slot to skip post instrumentation
baseTramp1+49:    pushl %ebp        // setup stack frame
baseTramp1+50:    movl %esp,%ebp
baseTramp1+52:    subl 0x80, %esp   // allocate virtual registers
baseTramp1+5:     pusha             // save registers
baseTramp1+44:    pushf
baseTramp1+45:    jmp <baseTramp1+60> // slot for jump to minitramp
baseTramp1+50:    popf              // restore registers
baseTramp1+51:    popa
baseTramp1+52:    leave             // undo stack frame

// relocated extra instructions from after the point go here
// there are no extra instruction from after the point in this case

// return to user code
baseTramp1+60: jmp <f+30>
```

The base tramp for the return point is similar to the base tramp for the entry point, except that the extra instructions added to the point, the `jl`, the `subl` and the `leave`, which were taken from before the point, are relocated to the beginning of the tramp. In this example, if the `jl` instruction branches, no instrumentation code will be executed.

The following example shows a minitramp for a `startWallTimer` primitive:

```
minitramp:        movl $0x8044f390,0xfffffffc(%ebp) // load timer address
                                                     // in virtual register
minitramp+7:      pushl 0xfffffffc(%ebp)           // push argument
minitramp+10:     movl $0x80585a4,%eax             // load function address
minitramp+15      call *%eax                       // call startWallTimer
minitramp+17:     addl $0x4,%esp                   // pop argument
minitramp+23:     movl %eax,0xfffffffc(%ebp)       // store result
minitramp+26:     jmp <baseTramp>
```

The references to `0xfffffffc(%ebp)` are references to a virtual register. (We are not doing code optimizations, though there are many opportunities to optimize this code.)

7 LINUX PORT

Inferior process modification and information through `ptrace` *and* `/proc` `[dyninstAPI/src/linux.C]`

The first major difference in Linux from Solaris is that the `/proc` interface doesn't support many of the process control features. The Linux `/proc` filesystem is a generally read-only setup, with most files simply providing information about the process in a text format.

Within `/proc`, there is a directory for each process, rather than a file on Solaris[2]. Each directory contains different files for different pieces of information about the process. In Solaris, each file contains the process' memory space, and IOCTLs on that file are used to gather other information and control that process.

`/proc/*/mem` contains the process' memory space, but it is currently read-only, due to concerns about the possibility of overwriting kernel memory in corner cases.

`/proc/*/stat` contains a list of numbers in ASCII format, space-delimited. Used information includes the process state (that is a char), and the process CPU times.

`/proc/*/maps` contains a list of mapped regions in the process memory space, along with device number and inode number, if the region is a file mapped to memory. This is especially useful in finding shared libraries which are loaded into memory.

`/proc/*/exe` is a link to the executable file for the process.

See 'man proc' and `/usr/src/linux/fs/proc` for further information.

Instead of using `/proc` to control and modify the process, we use the older `ptrace` interface.

For reading from the process memory space, we first try to simply read from `/proc/*/mem`, and if this fails, we use `ptrace(PTRACE_PEEKTXT, ...)` which reads a single word from the process at a time. Therefore, we must implement a function which reads a word at a time from the process, realigns the words, and re-packs them into the proper memory location in the parent process.

For writing to the process memory space, we use `ptrace(PTRACE_POKETXT, ...)` and write the data one word at a time, properly realigned to the addresses in the inferior process.

2. Is this still true for Solaris7?

To obtain the registers from the inferior process, we use `ptrace(PTRACE_GETREGS, ...)` and `ptrace(PTRACE_GETFPREGS, ...)` which write the registers to a buffer.

To change the registers in the inferior process, we use `ptrace(PTRACE_SETREGS, ...)` and `ptrace(PTRACE_SETFPREGS, ...)` which write the registers from a buffer to the inferior process.

NOTE: The register `ptrace` commands are only available in linux-2.0.35 and higher. We no longer support older versions of linux-2.0.x!

To obtain the actual state of the inferior process (running, stopped, etc.), we read from the `/proc/*/stat` file. The third space delimited field is a character which specifies the status. 'R' is running, 'T' is stopped, etc.

To wait on the inferior process for signals, we use `waitpid`, which simply waits until the inferior process receives a signal. We then check to see if the signal is one we should deal with. If it is not, the signal is forwarded back to the process using `ptrace(PTRACE_CONT, ...)`, the last parameter of which is the signal to send to the inferior process.

To continue the inferior process, we use `PTRACE_CONT` again with no signal.

To stop the inferior process, we simply use `kill(SIGSTOP)`.

To obtain the CPU time of the inferior process, we read the values for the inferior process user and system CPU time from `/proc/*/stat`, and do the proper arithmetic. The values are in ticks, or timeslices, which on a standard x86 Linux system occur at a rate of 100/second. This is checked for, however, through a one-time piece of code which finds the system idle time in ticks and in seconds and figures out the ticks/second.

Handling shared libraries in the inferior process `[dyninstAPI/src/linuxDL.C]`

The process for handling shared libraries in the inferior process is very similar to the process used on Solaris. The main difference is the problem of finding the **ld.so** library, which handles all of the other shared libraries. On Linux 2.0.x systems, the `/proc/*/maps` file shows all of the mappings, along with device number and inode number information, but there is no way to find the file from this information directly. Therefore, there is no way to tell which file is **ld.so**. The method used is to search the expected directory for a file matching the pattern "`ld*.so`", finding its device and inode number, and comparing it against each mapping. Then, the shared library handling can continue by the ELF method used in Solaris. On Linux 2.2.x and higher, however, the `maps` files also contains a path to each shared library. In this case, the pattern "`ld*.so`" is checked against these, and the file is found much more easily.

Inserting a shared library into the inferior process

`process::dlopenDYNINSTlib [dyninstAPI/src/linux.C]`

In order to insert a shared library into the inferior process, we depend on inserting code into the inferior to call `dlopen` on our library. This works well on Solaris, and for some programs on Linux. However, the version of **libc** currently used on Linux (**glibc 2.x**) does not include the public interface to `dlopen`. Instead, a separate library called **libdl.so** is used. If we insert code to call `dlopen` into a program not already linked to **libdl.so**, it will not work. Fortunately, the internal `_dl_open` function is available in all Linux programs which are dynamically linked. By inserting code to call this function instead, we can assure compatibility with all dynamically-linked Linux programs.

To deal with differences in **glibc**, we search for the `__libc_version` symbol, which contains a version string. If the string matches a known version of **glibc**, we work with that version. If the string is not found, or the version is unknown, we use the 2.0.x method.

In **glibc 2.0.x**, `_dl_open` takes the same parameters as `dlopen`, and the process is as simple as changing the name of the function to call.

In **glibc 2.1.x**, `_dl_open` takes an extra parameter of the modules which called `dlopen`. In this case, we need to provide this address, which is straightforward. Additionally, `_dl_open` uses a special function call convention because it is internal to **glibc**. Instead of pushing the parameters onto the stack, it passes all three in registers. To deal with this problem, we have to avoid the `AstNode` structure and generate a raw call and modify the registers for the parameters directly.

NB: This will probably change with each minor version change of **glibc**, and this code must be updated.

Inferior RPCs `[dyninstAPI/src/process.C]`

The majority of the usual method for executing inferior RPCs works fine on Linux. The problem is only in the checking for and dealing with the case where the inferior process is within a system call. It is dangerous to simply change the location to which the system call will return (which is the most simple approach), as this can corrupt the return value from the system call. Using `PTRACE_SYSCALL` seems promising, but this call traps at the entry and exit of the next system call, and so it would need to be used for every system call in the program, rather than just the current one: this is grossly inefficient. Instead, we simply find the location the system call will return to, and set a trap (or illegal instruction, actually) there. When this is hit, we restore the original code, save the registers, and move the process to the code we wish to execute. Since the registers were save *after* the system call instead of *during* it, the return value is safe.

Paradyn front-end threading package [libthread]

The threading package in Paradyn makes use of `setjmp` and `longjmp`. This is not generally a problem, except that we use a function pointer to the appropriate `setjmp` and `longjmp` functions on that platform. In Linux, `setjmp` is simply a macro to `sigsetjmp`, with the additional parameter specificied. This necessitates changing the threading package to use a macro for `setjmp` in the Linux case, as a function pointer simply will not work.

8 RUN-TIME INSTRUMENTATION LIBRARY

The run-time instrumentation library (`rtinst, libdyninstRT`) contains auxiliary functions and data for dynamic instrumentation. It contains functions to get wall and process time used by a process, to start and stop metric timers, to sample timers and counters, to report values to the Paradyn daemon, to report resources (such as message tags), and to report that a process is forking or doing and exec.

When an application process starts, it receives a signal that is caught by the Paradyn daemon (this signal is set up by `ptrace` or `/proc` file system calls). At this point, the daemon inserts the initial instrumentation in the application process. The initial instrumentation consists of inserting calls in some functions and system calls to call initialization and termination functions, or to reports events of interest, such as new resources, a fork, or an exec. The following functions are instrumented:

`main`: call to `DYNINSTinit()` and the entry point of `main`, and `DYNINSTexit()` at the return point.

`exit`: call to `DYNINSTexit()` at the entry point.

`fork`: call to `DYNINSTfork()` at the return point

`execve`: call to `DYNINSTexec` at entry point, call to `DYNINSTexecFailed()` at return point.

`pvm_send`: call to `DYNINSTrecordTag()` at entry point.

`DYNINSTsampleValues`: call to `DYNINSTreportNewTags()` at the entry point.

The function `DYNINSTinit()` is called at the start of the application process to initialize the run-time instrumentation library. Its main function is to set an alarm that sends a signal to the application process periodically. The alarm handler, `DYNINSTalarmExpire()`, is responsible for calling the functions to sample timers and counters and report the values to the Paradyn daemon. It also calls `DYNINSTreportBaseTramps()` to report the cost of instrumentation.

Enabled timers and counters are sampled by a call to `DYNINSTsampleValues()`. This is an empty function, but it is instrumented each time a metric is enabled, so that timers or counters are sampled when this function is called. The code that is inserted calls either `DYNINSTreportTimer()` or `DYNINSTreportCounter()` to read the timer or counter. The values are reported through a pipe that is created when the application is started by the Paradyn daemon, or by a stream socket that is created after the application has forked.

New dynamic heap segments are allocated in the application process by calling DYNINSTos_malloc. Section 5.7 describes dynamic heaps in detail.

Other functions of `rtinst` are called to report new resources, such as message tags (`DYNINSTreportNewTags()`), and to handle fork and exec by an application (`DYNINSTfork()` and `DYNINSTexec()`).

9 MDL IMPLEMENTATION

The Metric Description Language is used to specify what performance data to collect, and where. For the language specification, see the ***Paradyn User's Guide***. For a good high-level description of the implementation techniques, see the paper "*MDL: A Language and Compiler for Dynamic Program Instrumentation*" (Hollingsworth et. al.). The purpose of this section is to provide a better understanding of the MDL code, describing features and issues that are not documented elsewhere, and for providing a complementary and hopefully better reference than the code itself.

The MDL code consists of two parts: the front-end Paradyn process and the back-end Paradyn daemon. The front-end MDL code does lexical analysis, syntax analysis, and some type checking (which is part of the semantic analysis in the parlance of programming languages); the back-end does the rest of semantic analysis and intermediate code generation. The reason for the semantic checking being done by both the front-end and the back-end is due to the feature of dynamic instrumentation: the decision about what to instrument is deferred until after execution starts. Therefore, there are certain things that the front-end cannot check and must be relegated to the back-end. An example is an MDL expression containing a function call. The front-end can only check that the arguments of the call are valid MDL expressions and that the function call is used in valid syntactic context; whether the function exists in the application and is instrumentable can only be checked by the back-end MDL. However, the idea is to push static checking as much as possible into the front-end, so that errors can be caught early before the metrics are specified at runtime. Flex and Bison are used for lexical and syntax analysis.

The intermediate code generation is the process of translating a piece of MDL code into a DAG of `AstNode`s (see Section 5.4). The code generation is the process of translating the `AstNode`s into trampolines and inserting them into the application. This section does not describe the code generation of MDL, which is part of Paradyn's dyninstAPI (see the ***Dynamic Instrumentation API Guide***). We first list the important files of the MDL implementation. We then go through each stage of the analyses. At the end of this section we give a short reference list of the definitions of some frequently seen C++ classes in the MDL implementation.

9.1 Important files

Figure 13 lists the most important files in the MDL implemetation together with brief descriptions.

`paradyn/h/dyninstRPC.I`	An igen file containing class definitions for all of the MDL components such as Metric, Constraint, Statement, and Expression. Used by both the front-end and the back-end.
`paradyn/src/met/mdl.h`	Constant definitions and definitions for classes `mdl_var`, `mdl_env`. If you see some constants with all upper case letters while reading the MDL code, chances are that they are `#define`'d in this file. An `mdl_var` is an MDL variable, and the `mdl_env` is a repository of `mdl_var`s. You can think of `mdl_env` as the symbol table of MDL plus some methods. The MDL variables are collected into the static data member `mdl_env::all_vars`. MDL variables are pushed into `mdl_env::all_vars` when their scopes are entered, and popped out when their scopes are exited. Used by both the front-end and the back-end.
`paradyn/src/met/globals.h`	This file contains the declaration of global variables that both the front-end and the back-end need access to. The global variables include all MDL metrics, constraints, and resource lists. These MDL components are collected during the syntax analysis phase. Used by both the front-end and the back-end.
`paradyn/src/met/metScanner.l`	The input file to Flex for lexical analysis. All tokens and keywords can be found here. Only used by the front-end.

Figure 13: Crucial MDL files

`paradyn/src/met/metParser.y`	The input file to Bison for syntax analysis. Contains the entire MDL grammar, and hence is the definitive reference for the syntax and for determining whether some features are (should be) supported. Metrics, constraints, statements, etc. are created as part of the parse/grammar actions and collected into the global repositories declared in `globals.h`. Only used by the front-end.
`paradyn/src/met/mdl.C`	Type checking and `apply()` functions. See Section 9.2. Only used by the front-end.
`paradynd/src/mdl.C`	The major file of the back-end of MDL. Semantic checking and intermediate code generation. Only used by the back-end.
`paradynd/src/metric.C`	The definition of the class `metricDefinitionNode`, which describes metric instances. There are two types of node: aggregates and non-aggregates. For the aggregates, an `metricDefinitionNode` contains a vector of other `metricDefinitionNode`s, for non-aggregates, a node contains a vector of `dataReqNode`s. Only used by the back-end.

Figure 13: Crucial MDL files

9.2 Lexical and syntax analysis

Lexical and syntax analysis are done by the Paradyn front-end. The associated files are under the directory `paradyn/src/met`. It is important to be familiar with Flex and Bison before reading `metScanner.l` and `metParser.y`, and it is a good idea to get familiar with these two files, or the parts that you are interested in, before going on to others.

We do not explain the details of the files here[3], as the code itself serves exactly that purpose. Here we only point out some of the interactions among the files to help navigate.

The scanning and parsing of the configuration files starts from the routine `metMain()` in `metMain.C`. This routine calls `open_N_parse()` that calls the Bison function `yyparse()`, which in turn triggers the scanner and parser actions in `metScanner.l` and `metParser.y`[4]. Files `meClass.C` and `metParse.h` are support files for the scanner and parser, for example; they contain the definition of `struct parseStack`.

A bulk of the work done by the Paradyn front-end is type checking, which is done after Flex and Bison have already disected and collected all the syntactic parts. In the code, this occurs in

3. Those who do not need to know the details of MDL implementation, yet have to consult `metParser.y` for MDL grammars, may wonder what the symbols $$, $1, $2, etc. mean in `metParser.y`. $$ represents the left-hand-side of the rule, and $*i* represents the *i*th component on the right hand side of the rule, with *i* starting from 1. The type of those $-symbols is `struct parseStack` as specified by the line `#define YYSTYPE struct parseStack` in both `metScanner.l` and `metParser.y`.

4. In fact, Flex functions are called by Bison functions.

`metMain.C`'s `mdl_apply()`, after `open_N_parse()` is done. The type checking is implemented in routines with a heavily overloaded name: `apply()`. Many developers consider the `apply()` functions one of the most difficult to understand parts of the MDL implementation, probably because there are so many of them—not only in the front-end, but also in the daemon—and each `apply()` does different things. For a good grasp of those functions, we need a clear picture of the corresponding C++ classes and their relationships. File `paradyn/src/met/dyninstRPC.I` is the place to look for the class definitions of those syntactic components such as `mdl_metric`, `mdl_constraint`, `mdl_stmt`, etc. Let's use an example to show how `apply()` functions work.

Below is a metric called "procedureCalls" taken from `config/paradyn.rc`.

```
metric procedureCalls {
       name "procedure_calls";
       units operations;
       unitStyle unnormalized;
       aggregateOperator sum;
       style EventCounter;
       flavor = { winnt, unix, cow, pvm, mpi };

       constraint procedureConstraint /Code/* is replace counter {
             prepend preInsn $constraint[0].entry
             (* procedureCalls++; *)
       }
       constraint moduleConstraint /Code is replace counter {
             foreach func in $constraint[0].funcs {
                   prepend preInsn func.entry (* procedureCalls++; *)
             }
       }
       base is counter {
             foreach func in $procedures {
                   append preInsn func.entry constrained
                   (* procedureCalls++; *)
             }

       }
}
```

We draw a tree (Figure 14) to show the action of the parser. The tree also reflects the syntactic structure in this metric, with the relationship of the parent and children of the "nodes" being a *containing* relationship (e.g, procedureCalls metric *contains* a base statement and two constraints). We number each node to make the exposition clearer. Shown in parentheses are the actual C++ classes implementing the components.

The way `apply()` member functions work is essentially a pre-order visit of the tree starting from the root. `mdl_metric::apply()` (node 1) gets called, which would call the `apply()` member function on the statements in the base part of the metric (node 2), then `mdl_for_stmt::apply()(node 3)`, which in turn calls `mdl_instr_stmt::apply()(node 4)`, and then `mdl_icode::apply()(node 5)`, etc. After the subtree rooted at `node 2` is done, the subtree rooted at `node 7` is visited, and so on until the whole tree is "applied", and `mdl_metric::apply()(node 1)` returns.

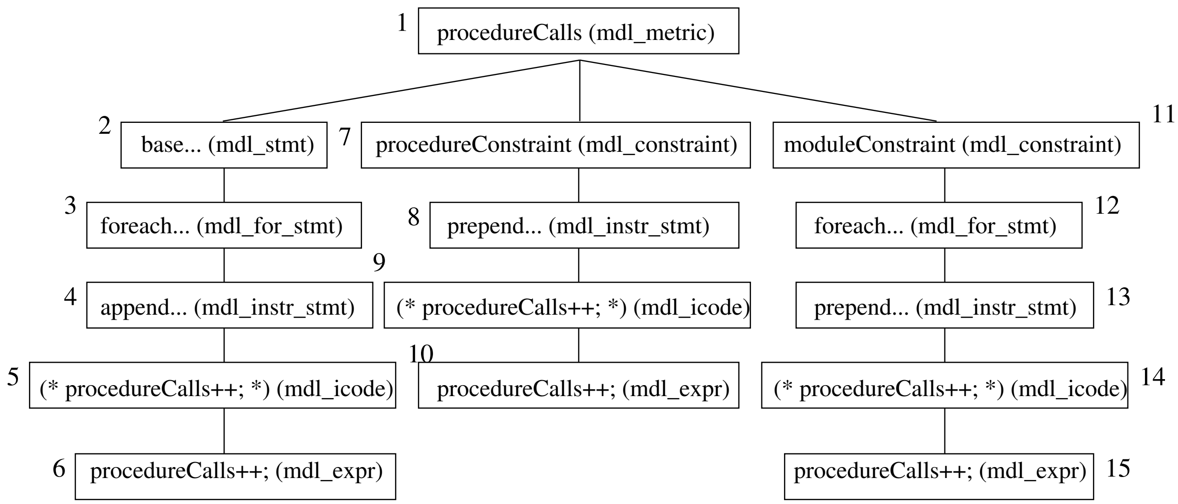


Figure 14: An example demonstrating how `apply()` functions work.

While visiting the tree, different checks are done inside `apply()` depending on which object the function is invoked. For instance, when `mdl_v_expr::apply()` is invoked on the expression `procedureCalls++`, it checks that `proceduceCalls` is of valid type (integers or counters).

At run-time, the sequence of `apply()` member functions starts from the `mdl_apply()` in `metMain()`, in the file `metMain.C`.

9.3 Semantic analysis and intermediate code generation

Semantic analysis and intermediate code generation are done by the back-end of Paradyn. This part of MDL comprises a few files under `paradynd/src`, with the major one being `mdl.C`. The two `mdl.C`'s (one in the front-end, one in the daemon) use the same class definitions in `paradyn/h/dyninstRPC.I`. In other words, the C++ classes for MDL metrics, constraints, statements, etc. encapsulate a superset of the functionalities needed for both the front-end and the daemon. Because of this, we can see some dummy function definitions in either file, since all definitions must be present to pass the compiler, even though they may not actually be used.

Due to the above reason, and also for symmetry, the semantic analysis and intermediate code generation of the daemon are also implemented with the hierarchies of `apply()` member functions. Again, the code is executed in the same pre-order-visit tree-like fashion as in the example of Section 9.2, with the exception of replace constraints as we will explain in a moment. This time, we generate intermediate code instead of mere checkings inside each `apply()`. For instance, for the expression `procedureCalls++` (node 6 in Figure 14), `mdl_v_expr::apply(AstNode*)` is called, and we generate an `AstNode*` as a result of evaluating this expression. After node 1 is successfully applied, a `metricDefinitionNode` is generated.

In the front-end, the syntax analysis is done on every syntactic component in the configuration file, yet the intermediate code and trampolines are only generated for those components that are actually used. For our example, if the metric `procedureCalls` is not enabled with a focus, it would not get processed by the back-end, and no intemediate code or trampolines would be generated for it. Furthermore, the two replace constraints (see the ***Paradyn User's Guide*** for a description of replace constraints) `procedureConstraint` and `moduleConstraint` would be applied only if their match paths (`/Code/*` and `/Code` respectively) match the focus. If neither one matches the focus, neither would be applied; if one matches the focus (note that there would be at most one match), the instrumentation statements inside the matching replace constraint are applied, and the instrumentation statements in the base of the metric are not. This is one exception in our `apply()` tree visit in our example: some subtrees may not be visited, hence applied, at all.

When the back-end receives a request to enable a metric-focus pair, it uses the MDL to generate intermediate code in the form of a DAG of `AstNode`s. The `AstNode`s specify what code to generate for the metric-focus pair. For each metric-focus pair a `metricDefinitionNode` is created (in `createMetricInstance()` in `paradynd/src/metric.C`). The `metricDefinitionNode` contains the DAG of `AstNode`s and the information about where the generated instrumentation code (trampolines) should be inserted. After the `metricDefinitionNode` is successfully created, the trampolines are generated and inserted into the application executable, see `AstNode::generateCode()` and `AstNode::generateCode_phase2()` (code generation is part of dyninstAPI).

9.4 Where these classes are defined

The classes below are important to the MDL implementation. Although tools like ctags/etags can be used to pinpoint their definitions, they are listed here just for reference. (This is a very short list, hope to add some more).

Class	**Where it is defined**
`AstNode`	`dyninstAPI/src/ast.h`
`dataReqNode`	`paradynd/src/metric.h`
`function_base`	`dyninstAPI/src/symtab.h`

`instPoint`	`dyninstAPI/src/instPoint-power.h`
	`dyninstAPI/src/instPoint-sparc.h`
	`dyninstAPI/src/instPoint-x86.h`

`resource`	`paradynd/src/resource.h`
`metricDefinitionNode`	`paradynd/src/metric.h`

Figure 15: Important MDL classes.

10 IGEN INTERFACE GENERATION

10.1 Overview of Igen

Igen automates the creation of remote interfaces. Interfaces are like remote procedure calls, but the endpoints (client and server) can be threads or processes. Igen supports generation of RPC calls using either threads or XDR (or PVM?) as transport.

10.1.1 Synopsis

```
igen -xdr | -thread | -pvm [ -header | -code ] <spec>.I
```

10.1.2 Output

The `<spec>.I` file specifies the interface template to use to generate the source and header files. All generated files will use <spec> in their name:

`<spec>.C` – bundlers for the types that can be passed.

`<spec>.CLNT.C` – client side code for users of the interface.

`<spec>.SRVR.C` – server code for providers of the interface.

`<spec>.h`, `<spec>.CLNT.h`, `<spec>.SRVR.h` – class headers.

Note that member functions declared in `<spec>.SRVR.h` are not generated by Igen, except for the class constructor and mainLoop. These functions are called by the server when it receives a request from the client. These functions must be provided by the programmer.

10.1.3 Memory

Igen frees all memory that it allocates, with one exception. Return types in the client code may be a structure or an array class. The memory allocated for these return types will not be deallocated by Igen.

10.1.4 Upcalls

Upcalls from the server to the client are supported, however, they will only be seen when the client is waiting for a response from a synchronous call to the server. There is a way to force the client to attempt to handle an upcall. The client has a member function awaitResponce which will handle any upcall requests that exist, but awaitResponce will block. The file descriptor should be checked to see if it is ready for reading before calling awaitResponce.

10.1.5 Interface template

An interface looks like:

```
$remote <interfaceName> {
        $base <int>;
        $version <int>;
        $virtual [$async | $array] <member function definitions>
        $virtual $upcall [$async] <member function definitions>
        $cmember type variable;
        $smember type variable; }
```

The $array keyword causes igen to genarate an array class and use this as the array type. The class has a member specifying the size of the array and a pointer to the data.

The $virtual keyword causes the igen generated functions to be declared virtual. For upcalls, the client function is declared virtual. For non-upcalls, the server function is declared virtual.

The $smember and $cmember keywords cause igen to put the type and variable declaration into the client or server class. $smember specifies that the server class is to include the type and variable as a public data member. $cmember specifies that the client class is to include the type and variable as a public data member.

The $base keyword defines the first message tag to use for creating request and responce message types. Since TAGS should be unique to an application, this value should not confilct with other interfaces that might get linked into the same process.

The integer after the keyword $version indicates the protocol version of this interface. For XDR based protocols this version is verified when the client and server rendevous. For thread based interfaces, Igen relies on the fact that changes to an interface generally change the signature of at least one function in the interface, and that version incompatabilities should be resolved by the C++ linker in that case.

The member functions are the basis of the interface. A provider of an interface defines the member functions in the class <interfaceName>. Igen generates a shadow class <interfaceName>User with the same member functions. The <interfaceName>User member functions are really RPC style stubs that invoke the remote member functions.

The $upcall keyword permits interfaces to support upcalls. Upcalls are a way for an interface to indicate to its user that an "interesting" event has occured. Upcalls are by default synchronous, but can be made asynchronous by adding the keyword $async after the keyword $upcall.

The $async keyword placed before a function definition prevents igen from generating a wait for reply after make the remote procedure call. No reply will be made by the receiver of the remote procedure call.

10.2 Igen grammar

[Words in lowercase are nonterminals; words with punctuation in them (e.g., $), surrounded by quotes, and in all CAPITALS are terminals.]

```
completeDefinition   -> parsableUnitList
                      | error
parsableUnitList     -> parsableUnitList parsableUnit
                      | lambda
parsableUnit         -> interface_spec
                      | typeSpec
interfacePreamble    -> interfaceName { interfaceBase interfaceVersion
interface_spec       -> interfacePreamble definitionList } ;
interfaceName        -> IDENTIFIER
interfaceBase        -> $base UNSIGNED_INT_LITERAL ;
interfaceVersion     -> $version UNSIGNED_INT_LITERAL ;
forward_spec         -> 'forward' IDENTIFIER ;
definitionList       -> definitionList definition
                      | lambda
optUpcall            -> $virtual
                      | $async
                      | $virtual $async
                      | $upcall $async
```

```
                   | $virtual $upcall $async
                   | lambda
optFree           -> $free
                   | lambda
optRef            -> &
                   | lambda
definition        -> optFree optUpcall optConst typeName pointers optRef
                              IDENTIFIER ( arglist ) ;
                   | $cignore[^$]*$cignore
                   | $signore[^$]*$signore
optIgnore         -> $ignore[^$]*$ignore
                   | lambda
optAbstract       -> 'abstract'
                   | lambda
classOrStruct     -> optAbstract 'class'
                   | 'struct'
typeSpec          -> classOrStruct IDENTIFIER optDerived {
                   fieldDeclList optIgnore } ;
optDerived        -> : IDENTIFIER
                   | lambda
fieldDeclList     -> fieldDeclList fieldDecl
                   | lambda
fieldDecl         -> optConst typeName pointers IDENTIFER ;
typeName          -> IDENTIFIER
                   | IDENTIFIER : : IDENTIFIER
                   | IDENTIFIER < typeName pointers >
optConst          -> 'const'
                   | lambda
pointers          -> * pointers
                   | lambda
funcArg           -> optConst typeName pointers
                   | optConst typeName pointers IDENTIFIER
                   | optConst typeName & IDENTIFIER
                   | optConst typeName &
nonEmptyArg       -> funcArg
                   | nonEmptyArg , funcArg
arglist           -> nonEmptyArg
                   | lambda
```

11 MAKEFILE ISSUES

11.1 Overview of Makefile organization

The files `make.config`, `make.program.tmpl`, and `make.library.tmpl` (located at the root of the Paradyn source code tree) are the basis for compiling the Paradyn system. They define generic rules and Makefile dependencies, and are flexible enough that most `Makefile`s for Paradyn system components are kept short and simple. The shadow files **n**`make.config`, etc., are similar and only required by *nmake* on Windows NT; they are also simpler, supporting only that one platform.

A Makefile for a given platform (such as SPARC/Solaris) and given program (such as Paradyn, Paradynd, or Igen) is typically organized as follows. Several Makefile variables are first defined; for example, you will see lines such as `USES_TCLTK=true` and `USES_FLEX=true` in the Paradyn platform Makefiles[5]. Then the Makefile executes the line `include ../../make.config`, which reads in the file `make.config`. This file defines default dependencies, default compiler flags, library paths, include directories, and so on. At many points, `make.config` will check to see if certain Makefile variables (such as `USES_TCLTK` and `USES_FLEX`) are defined; if so, it performs additional tasks.

For example, if `USES_TCLTK` is defined, then `make.config` sets the `TCL2C` makefile variable to the appropriate path (for when the `tcl2c` script is run), adds the path to the Tcl/Tk include files to the compiler flags, and adds the path to the Tcl/Tk libraries to the compiler's library-search path.

After `make.config` is read in, the Makefile may make a few changes to the Makefile variables, as `make.config` has assigned them. For example, the line `CXXFLAGS += -O3` would make C++ compile its files with the highest level of optimization (because the Makefile variable `CXXFLAGS` is in turn used by GNU make when compiling C++ files).

Next, a Makefile should have the line `include ../make.module.tmpl`. This file is the platform-independent part of the module build (just as the `Makefile` is the platform-specific part). Its function is to set Makefile variables that will be used by `../../make.program.tmpl`, which is included next. The most important of these Makefile variables are `TARGET` (which specifies the name of the final binary that the linker should write to)[6], `SRCS` (which specifies the source files), `LIBS` (which specifies additional libraries not automatically defined by `make.config`), and `SYSLIBS` (similar to `LIBS`, but intended for non-Paradyn libraries). Note that these Makefile variables are usually appended to, as opposed to overwritten.

For example, we see the line `LIBS += -lpdutil -lpdthread` in the `make.module.tmpl` for Paradyn, instead of the line `LIBS = -lpdutil -lpdthread`. This is important, because typically, `make.config` will already have defined some initial values for these Makefile variables, which should be appended to, rather than overwritten.

Makefiles for libraries (such as VisiLib) follow a similar approach; the major difference is that at the last step `../../make.library.tmpl` is included instead of `../../make.program.tmpl`.

---

5. Note that most Makefile tests concern whether something is defined (`ifdef`), and therefore *any* non-empty definition is equally considered: be careful to comment-out or undefine undesired definitions rather than ineffectively setting `USES=false` (which will still be considered defined!). For consistency, `true` is the prefered definition when one is required.
6. Some modules require to (sometimes) build/install multiple `TARGETS` or an alternative `ALT_TARGET`.

Note that there are features used in the make configuration files that are specific to the GNU version of make (we currently use version 3.74) and may not be understood by other makes.

11.2 Site-dependency issues

While the top-level `Makefile` is (Unix) system-independent), the file `make.config` will need to be edited to conform to your system's configuration. For example, the path to your Tcl/Tk library, the path to your flex library, and the path to X-Window's include files will likely differ from settings we have used. You should edit the `make.config` file and make the following changes:

- The destinations for installing Paradyn libraries and programs are specified, *relative to the core of the Paradyn source distribution*, by `LIBRARY_DEST` and `PROGRAM_DEST`. Note that while alternate locations may be specified, modification of the standard Paradyn build and install directory structure is not recommended. Whenever make is performed from the toplevel source (core) directory, a check is made to determine whether these directories already exist, an attempt is made to automatically create them. If this fails, or make is run directly from module subdirectories without these directories existing (and writable), the build will likely be unsuccessful, as it relies on installing and using components as they are built.
- Search for `BACKUP_CORE`, and replace its path with either "`../..`" or the location of the root of the Paradyn distribution (`PARADYN_ROOT`). Most sites will not need to use this variable; it specifies alternate locations to search in the event that the primary `TO_CORE` variable doesn't find that it was looking for. Search the `make.config` file for uses of `TO_CORE` and `BACKUP_CORE` to get an understanding of how they are used as (primary and secondary) directory prefixes.
- Search for the line `TCLTK_DIR` and replace the path with the location where Tcl/Tk has been installed on your system. Also check that the names of your Tcl and Tk libraries corresponds to those listed in `TCLTK_LIBS`: on some systems the libraries may be called `tcl8.3` and `tk8.3` instead of simply `tcl` and `tk`. More specifically, the directory `$(TCLTK_DIR)/lib` should contain `libtcl.a` and `libtk.a` (or the equivalent names specified by `TCLTK_LIBS`).
- Search for `FLEX_DIR` and change its value to the location where the flex library (`libfl.a` or `libfl.lib`) has been installed on your system.
- Depending on where the X-Windows include files have been installed on your system, you may need to tell the compiler where to find them. Search for `USES_X11` and observe the contents of what's already there. There are checks for different platforms and corresponding changes. For some platforms, nothing is done because in our configuration, the compiler doesn't need to be told where the X-Windows include files are, where the X-Windows libraries are, and so on. Depending on your system setup, you may need to make some changes here to `X11_LIB`, `X11DIR` and possibly others.
- Check that utilities, such as `YACC` (`bison`) and `PERL` (version 5 or later) are available with the names (and perhaps paths) specified.
- A private `make.config.local` file is read (if it exists) after `make.config` itself, and can be used to override general make configuration defaults. This is generally an appropriate place to (optionally) define `BUILD_MARK` and `BUILD_NUM` (build identifiers), etc.
- The `nmake.config` file for Windows NT's *nmake* is similar (and generally simpler) and should be modified as described above. One additional configuration option relates to the use

of Unix shell utilities (such as those freely available from Cygnus) or roughly-equivalent standard Windows NT commands: currently if *nmake* is run from a shell (and the `SHELL` environment variable is duly defined) then the more functional Unix utilities are used.

11.3 The `DEPENDS` file

The first time a program is compiled for a given platform (e.g., `paradynd/sparc-sun-solaris2.4`), the equivalent of the command `make depend` is automatically issued. It creates a file `DEPENDS` in the platform directory which contains header file dependencies for all of the source code files; these dependencies will automatically be included by `make.program.tmpl`. You can manually recreate this file (a good idea if you change the source code in such a way that you modify what `.h` files are included in one or more `.C` files) by typing `make depend`.

If the Makefile variable `EXPLICIT_DEPENDS` is *not* defined, then the make system will (for consistency) perform a `make depend` every time a source file changes. This can take a good bit of time, so you may wish to define `EXPLICIT_DEPENDS` in a platform-specific Makefile to avoid this (or you could define it in `make.config` to make it the default). Simply put the line `EXPLICIT_DEPENDS=true` in the appropriate location (before `make.program.tmpl` is included).

Note that automatic generation of `DEPENDS` files is not supported under Windows NT. The `DEPENDS` files must be manually updated as dependencies change.

11.4 Igen Files

Paradyn, VisiLib and Paradynd use Igen-language files (with the `.I` suffix) to define the remote procedure call interface between them. Whenever you change `.I` files, it is important to re-compile all Paradyn components which use them. To do this, type "`make clean`" followed by `make` in the appropriate platform directories for these programs.

11.5 Building on Windows NT

We are using the Visual C++ 6.0 compiler and Microsoft *nmake* programs to build Paradyn on Windows NT. Because the configuration and Makefiles used on other platforms are not compatible with *nmake*, there are a different set of configuration files for Windows NT called `nmake.config`, `nmake.module.tmpl`, `nmake.library.tmpl` and `nmake.program.tmpl`. Each file is the equivalent of the similarly named configuration file for the Unix platforms. To compile a module, go to the Windows NT platform directory in the module (`i386-unknown-nt4.0`) and type `nmake` (or `nmake install`). There is no top-level Makefile (the `core/Makefile` will not work with *nmake*), though the `scripts` directory contains both Unix shell (`make-nt.sh`) and command (`make-nt.bat`) scripts that will try to compile everything.

The following packages are needed to build Paradyn: bison, flex, Tcl/Tk, and ONC RPC (an implementation of Sun RPC). One of the include files in the ONC RPC package, `RPC/xdr.h`, needs to be modified to compile with the Visual C++ compiler. (We have these packages installed under `p:/paradyn/packages/winnt`, and this path should be updated as appropriate for your system. A gzipped tarfile of ONC RPC v1.12 with the `RPC/xdr.h` file already modified is available from `ftp://grilled.cs.wisc.edu/paradyn/etc/oncrpc112winnt.tar.gz`).

To run the Paradyn daemon on Windows NT, the dynamic link library `oncrpc.dll` must be in some directory that is listed on your `PATH` environrment variable, so that the Paradyn daemon can use Sun RPC calls to communicate with the Paradyn front-end.

12 MPI APPLICATION SUPPORT

Paradyn currently supports native MPI on AIX/SP and MPICH on x86/Linux and x86/Solaris. Metrics based on MPI library functions are defined in the usual way in the Paradyn configuration file (`paradyn.rc`). This section describes special support for starting MPI applications (i.e., distributed collections of MPI processes) under Paradyn control. Since application startup is not specified in the MPI standard, the mechanisms used by each implementation typically vary and Paradyn requires implementation-specific support for each case. In all cases, support is currently only available for creating/starting MPI applications under Paradyn control, rather than attaching to existing collections of MPI application processes.

12.1 MPICH Support

Paradyn includes support for MPICH applications on collections of workstations. The current implementation has several limitations which are given below.

- Cluster nodes should share a common file system with the host used to launch the application. For each MPI application being launched, the Paradyn frontend creates a startup file that should be accessible from all nodes in the cluster. In the future, the frontend may ship this file to other nodes via an rcp-like mechanism or use environment variables to avoid this need.
- Only x86/Linux and x86/Solaris platforms are currently supported, both as homogeneous and heterogeneous collections. A proper SPARC/Solaris implementation would require an ability to access function arguments off the stack (parameter 7 and higher).
- Paradyn requires MPICH version 1.2.0. Older versions of MPICH can be supported by re-linking an MPI application with the profiling library `libpmpich.a` and the Paradyn wrapper library `libpdmpich-1.1.0.a`.
- Paradyn currently does not support any MPICH drivers other than the default P4 driver, however, other drivers can be handled in a similar fashion. The pure shared memory driver`(shmem)` can be supported by enabling the default follow-fork instrumentation in the daemon. The same method may be sufficient for the mixed P4+shmem driver, however, the last driver performs several `exec()` system calls at startup which may not be handled reliably by Paradyn. A potential solution may be to allow `the exec()` calls to happen unnoticed, which should not involve many changes in the daemon.

12.1.1 MPICH job startup procedure

Consider an MPI application that is to be started on 3 nodes (A, B, C) via the following command: `mpirun -np 3 hello`. Figure 16 provides a step-by-step description of the default P4 driver startup procedure. In the diagrams, black/solid arrows indicate process creation, red/dashed arrows indicate communication and blue/dotted arrows indicate process control.Long dashed lines indicate machine boundaries. Figure 17 describes startup of the same `hello` application under

Paradyn. The core idea is to make MPICH start Paradyn daemons instead of the real application nodes. It is the daemon's responsibility to launch the application after that.

Figure 16: MPICH Job Launch Procedure

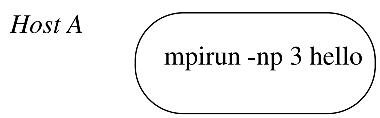
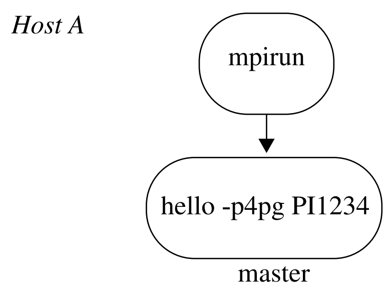
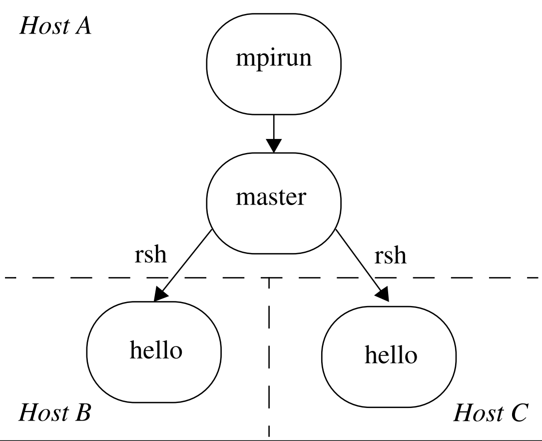
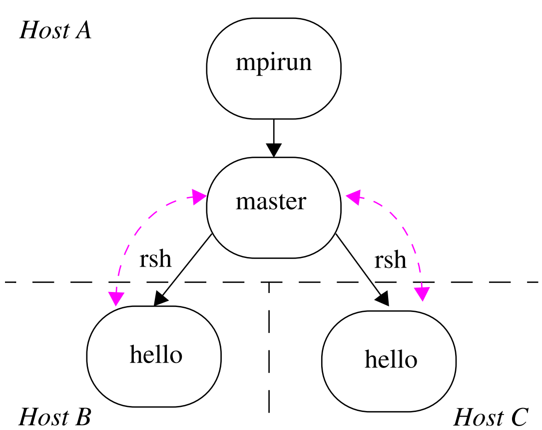
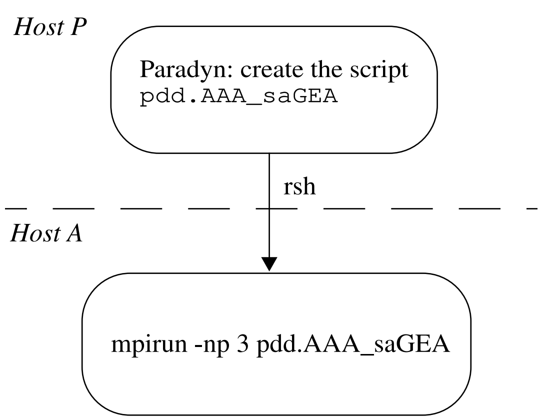
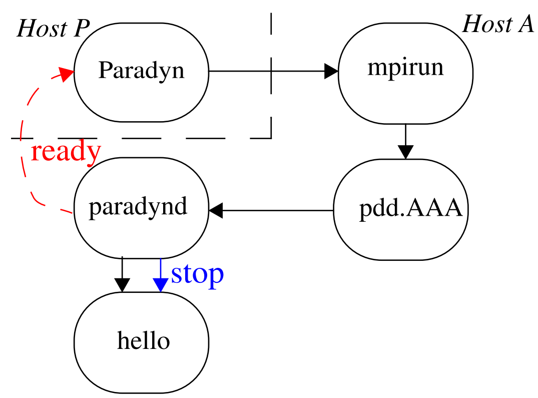
	The `mpirun` command is issued on Host A.
	The `mpirun` script performs certain preprocessing steps: creates a `procgroup` file with the three host names and sets up environment variables, if necessary. After that, it launches the first copy of the application locally, passing the `procgroup` file name (`PI1234`) as an argument. This program instance (the "master") plays a special role in the startup process, but it becomes an ordinary computational process once all nodes are started.
	The master starts running and hits the `MPI_Init()` function. This function analyzes the procgroup file and creates computational processes (slaves) on the other nodes (B and C). Actual process creation may happen either through an rsh-like program or through the special P4 daemon. Each slave is started with the master location (hostA, port#) passed through the command line: `hello -p4amslave hostA port#` Notice that user-supplied command line arguments are hidden from the slaves at this point.
	The slaves run until they hit the `MPI_Init()` function. This function analyzes the command line arguments and connects back to the master. The master ships the user-supplied command line arguments to the slaves, completes the initialization and all processes start running.

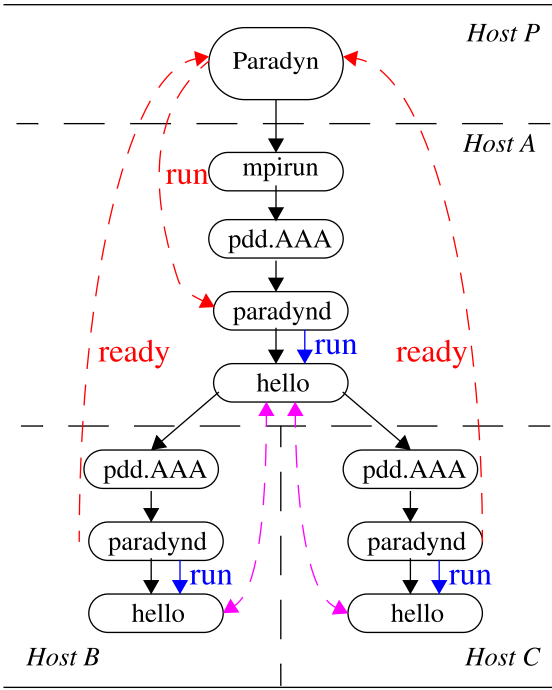
Figure 17: Paradyn MPICH Job Launch Procedure



The user enters a complete mpirun command in a Paradyn command file or in the Paradyn (Setup, Define a New Process) command field. Paradyn parses the command line, creates a helper script, replaces the program name with the script name in the command line and executes the command line on the specified target machine. The purpose of the script is to start paradynd with proper arguments (`frontend_host,frontend_port,prog_name`). These arguments can not be included in the user-specified command line, because MPICH hides it from slave processes (paradynd's) until they complete `MPI_Init()`.



`mpirun` proceeds as described in Figure 16 and starts the first pdd script. The script launches paradynd, paradynd starts the user application, stops it at the beginning of `main()` and communicates to the frontend.



The user hits the "`Run`" button and the application starts running. This instance becomes the master, so it attempts to create slaves, but runs our scripts on the remote hosts instead. The scripts create Paradyn daemons. The Paradyn daemons create inferior processes and communicate to the frontend. The frontend adds these processes to the resource hierarchy and tells paradynd's to continue them. Finally, slave processes hit `MPI_Init()`, notify the master of arrival and the MPI application starts running.

12.1.2 Supporting MPICH on other platforms

Most of the described infrastructure is platform-independent. To support MPICH on a new platform, one may not need to change the frontend. Following is a list of the required changes to the daemon. See `paradynd/src/init-linux.C` for details.

- Instrument `fork()` with the specialized `DYNINSTmpi_fork()` routine instead of the standard `DYNINSTfork()`. Currently, we do not need to follow `fork()` in MPI applications. The goal of the `DYNINSTmpi_fork()` routine is to perform cleanup after `fork()`.
- Do not instrument the `exec()` call.
- Invoke `instMPI()` to instrument several MPI functions with tag and group- recording code snippets.