Paradyn Parallel Performance Tools

Self-propelled Instrumentation

Developer's Manual

1.0b Release
September 2012

Computer Science Department
University of Wisconsin–Madison
Madison, WI 53711

Computer Science Department
University of Maryland
College Park, MD 20742

Email bugs@dyninst.org
Web www.dyinst.org

Contents

1 Introduction

Self-propelled instrumentation is a binary instrumentation technique that dynamically injects a fragment of code into an application process on demand. The instrumentation is inserted ahead of the control flow within the process and is propagated into other processes, following communication events, crossing host boundaries, and collecting a distributed function-level trace of the execution.

Self-propelled instrumentation contains two major components, *Agent* and *Injector*. *Agent* is a shared library that automatically inserts and propagates a piece of payload code at function call events in a running process, where the payload code contains user-defined logic, such as generating trace data for later inspection. The instrumentation would propel itself within the process by following control flow and across thread boundaries, process boundaries, or even host boundaries by following communication flow. *Injector* is a process that causes an application process to load the Agent shared library, where the Injector should have at least the same privilege as the application process. Self-propelled instrumentation does binary instrumentation within the application process's address space, avoiding use of the debugging interfaces (e.g., Linux ptrace and Windows debug interface) and costly interprocess communications. Therefore, self-propelled instrumentation does not add significant overhead to a process during runtime.

Self-propelled instrumentation can be used in many applications that require low overhead instrumentation and full automation of instrumentation propagation following control flow. For example, we have used self-propelled instrumentation for problem diagnosis in distributed systems [**?**] and for automated diagram construction for complex software systems in security analysis [**?**].

2 Abstraction

Self-propelled instrumentation has two major components, *Agent* that is a shared library injected into an application process's address space, and *Injector* that injects *Agent*. The following subsections describe the lower level components in Agent and Injector in details.

2.1 Agent

- Agent. It manages the configuration and does instrumentation. An Agent instance is created in the init function of the *Agent* shared library.
- Event. It specifies what kind of initial instrumentation should be done once the *Agent* shared library is loaded. Currently, there are three types of Event: 1) instrumenting all

callees in *main* function right away; 2) instrumenting all callees of specified functions right away; 3) instrumenting specified function calls right away.

- Payload function. It contains user-specified code. From user's perspective, a payload function will be invoke before or after each function call in the process. There are two types of payload functions, *entry payload* that is invoked before each function call and *exit payload* that is invoked after each function call.

- Point. It represents an instrumentation point at current function call and is used in Payload function.

- Control Flow Graph (CFG) structures. CFG structures include Object, Function, Block, and Edge. An Object represents a binary file (i.e., an executable or a shared library), and contains a set of functions. A Function contains a set of Blocks. A Block is a basic block. An Edge connects two Blocks. Users can get related CFG structures of current function call from Point.

- AddressSpace. It represents the address space of the process. It contains a set of Objects in the process. Also, it implements some memory management primitives used by the instrumentation engine.

- Parser. It represents a binary code parser that parses binary code into structural CFG structures, i.e., Object, Function, Block, and Edge.

- Propeller. It manages intra-process instrumentation propagation, where it finds function call Points inside current function and uses Instrumenter to insert Snippets at these points.

- Snippet. It represents a patch area that contains function calls to the Payload function and the relocated function call or the relocated call block.

- Instrumenter. It is the instrumentation engine that uses a set of Instrumentation Workers to insert Snippets to function call points.

- Instrumentation Worker. It represents a mechanism of installing instrumentation. Currently, four types of Instrumentation Workers are implemented: 1) relocating original function call instruction; 2) relocating original call block; 3) relocating nearby large springboard block; 4) using trap instruction.

- IpcMgr. It manages inter-process instrumentation propagation by creating Channels and using IPC Workers.

- Channel. It represents a unidirectional communication channel, containing local process name and remote process name.

- IPC Worker. It implements inter-process instrumentation propagation for a particular IPC mechanism (e.g., TCP, UDP, pipe).

2.2 Injector

Injector is provided as a command. There are two types of injections. One is to inject the *Agent* shared library at the very beginning of a process. The other is to inject the *Agent* in the middle of a running process.

The first type of Injector relies on dynamic linker (i.e., setting the environment variable LD_PRELOAD to the path of an *Agent* shared library). The second type uses ProcControlAPI to force an application process to invoke functions in the dlopen family.

3 How it works

This section describes how self-propelled instrumentation works in details. Each subsection is a major step in the workflow.

3.1 Building Agent

Users build their own *Agent* shared library using self-propelled instrumentation's API.

1. Coding. Users need to write two pieces of code: 1) payload function; 2) configuration code that registers payload function and does some customization and configuration. The configuration code must be executed right away when the *Agent* shared library is loaded into the application process, so the configuration code should be in the init function of the *Agent* shared library, i.e., the function with gcc directive __attribute__((constructor)).
2. Building. Users build the code into an *Agent* shared library linking with *libagent.so* provided by the self-propelled instrumentation infrastructure.

3.2 Injection

Users run *Injector* in command line. They specify in command line arguments the path of an *Agent* shared library and the application process to inject to.

One trick to check whether the *Agent* shared library is injected successfully is to look at memory maps file of the application process, i.e., /proc/PID/maps.

3.3 Initialisation

per diagram The initialisation code is executed right away when *Agent* shared library is loaded into the application process. It tells self-propelled instrumentation what are payload functions provided by users, how would initial instrumentation be done, whether or not to enable inter-process instrumentation propagation ...

3.4 Initial Instrumentation

Once the configuration code in the *Agent shared library* finishes execution inside the application process, the initial instrumentation would be performed right away, e.g., instrumenting all function calls inside the *main* function.

3.5 Instrumentation Propagation

When one of the initial instrumentation gets executed, then instrumentation propagates itself either within the process by following control flow, or across process boundaries by following communication flow.

3.5.1 Intra-process propagation

Instrumenting all the function calls inside *main* function internally calls the instrumentation engine, before each function call. Instrumentation engine, in turn, propagates the instrumentation to the called function. If the called function is within the same process, then instrumentation engine directly propagates the instrumentation to the called function: i.e it instruments all the function calls inside the called function. The instrumentation engine also executes payload function either before instrumenting or after instrumenting depending on whether it is an entry payload or exit payload function. After the instrumentation is done, the application's normal intra-process function call execution takes place. The workflow is visualized in Figure 1.

3.5.2 Inter-process propagation

Similar to intra-process propagation, inter-process propagation also instruments all the function calls inside an instrumenting function, but the called function lies in a different process or host. So the instrumentation engine, which gets called before every function call of the instrumenting function, identifies that the called function is an inter-process function. It then propagates the call to the SPI daemon of the appropriate host where

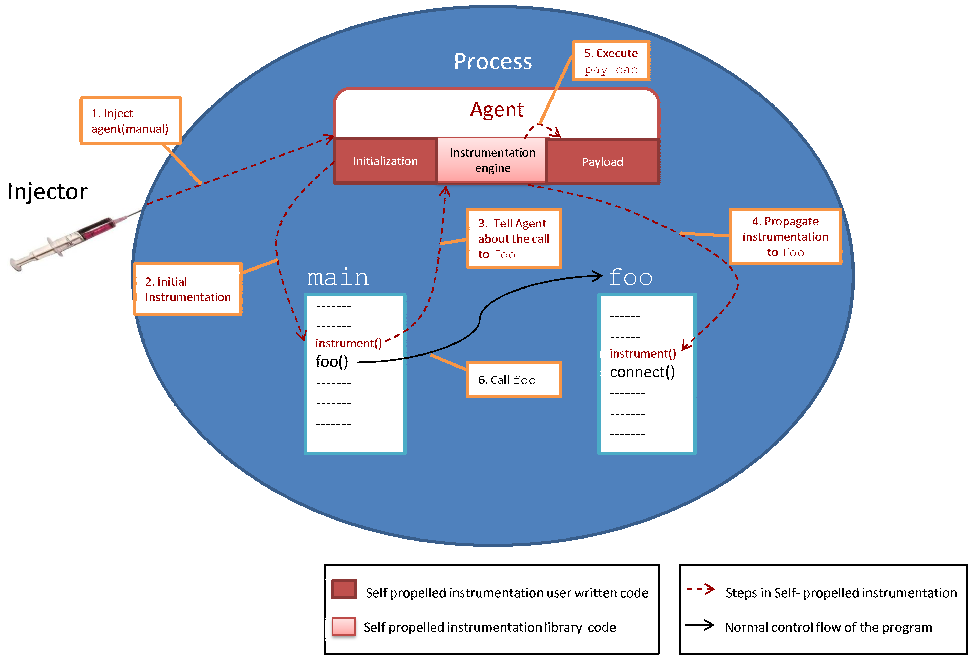

Figure 1: Intra-process Self-propelled Instrumentation Workflow

the called function is present. The called SPI daemon then injects the agent shared library automatically inside the process where the called function is located. Then the usual process of *initialisation* and *initial instrumentation* goes on inside the called function. After the instrumentation is done, the application's normal inter-process function call execution takes place. In the mean time, the instrumentation engine executes the payload depending on whether it is an entry payload or an exit payload. The workflow is visualized in Figure 2.

4 Examples

To illustrate the ideas of Self-propelled instrumentation, we present some simple code examples that demonstrate how the API can be used.

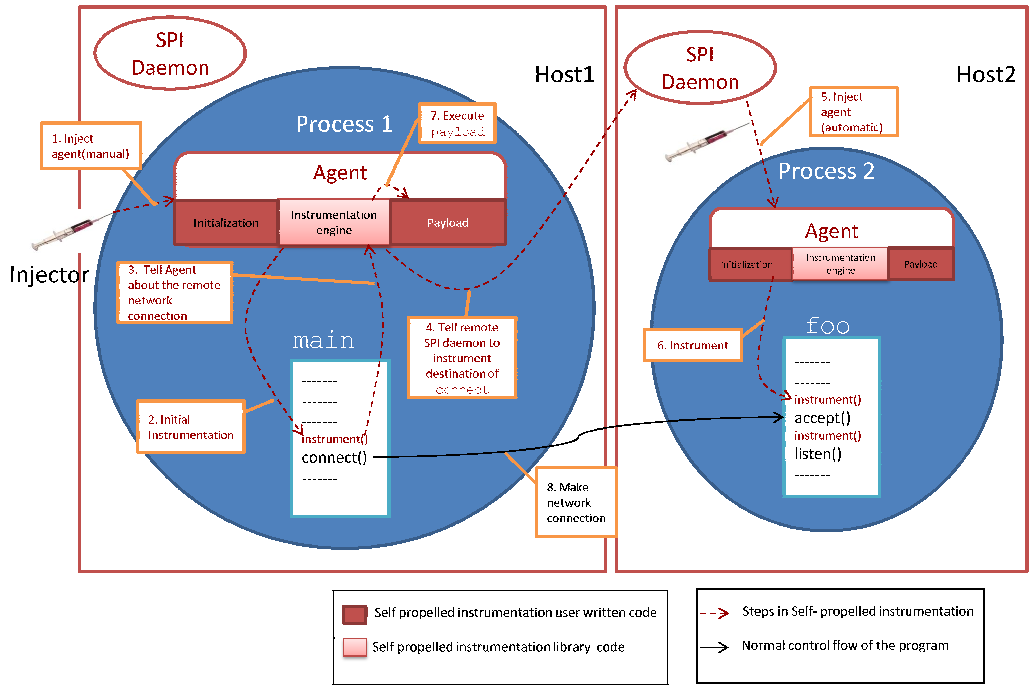

Figure 2: Inter-process Self-propelled Instrumentation Workflow

4.1 Writing Payload

The primary task for users using self-propelled instrumentation is write payload functions that are invoked before or after each function call.

Listing 1: Writing payload functions

```
1  // Entry payload function that is invoked before each function call
2  void entry_payload(SpPoint* pt) {
3    // Get function from point
4    SpFunction* func = sp::Callee(pt);
5    if (func == NULL) return;
6
7    // If we encounter the system call exit, then print its argument
8    if (func->name().compare("exit") == 0) {
9      // Get argument
10     sp::ArgumentHandle h;
11     int* exit_code = (int*)sp::PopArgument(pt, &h, sizeof(int));
12     printf("Exit Code: %d\n", *exit_code);
```

```
13    }
14    sp::Propel(pt);
15  }
16
17  // Entry payload function that is invoked after each function call
18  void exit_payload(SpPoint* pt) {
19    // Get function from point
20    SpFunction* func = sp::Callee(pt);
21    if (func == NULL) return;
22
23    // If we encounter the system call fork, then print its return value
24    if (func->name().compare("fork") == 0) {
25      pid_t child_pid = sp::ReturnValue(pt);
26      // Print child process id from parent process side
27      if (child_pid > 0) {
28        printf("Child process id: %d\n", child_pid);
29      }
30    }
31  }
```

In the above code, we illustrate some common operations that users may want to do in entry payload function or exit payload function. Users can get CFG structures (e.g., function, block ...) from the argument *pt*. Typically, there are some conditional branches to handle different functions, e.g., handling exit and fork in the above code. Users can also get arguments or return value for current function call.

4.2 Configuration

Users also need to provide the configuration code in the constructor function of the Agent shared library, which is mainly to register the user-provided payload functions.

Listing 2: Configuration code

```
1  __attribute__((constructor))
2  void MyAgent() {
3    sp::SpAgent::ptr agent = sp::SpAgent::Create();
4    // Register entry payload function
5    agent->SetInitEntry("entry_payload");
6    // Register exit payload function
7    agent->SetInitExit("exit_payload");
8    // Initiate instrumentation
9    agent->Go();
10 }
```

The above code shows the minimum operations needed to configure self-propelled instrumentation. The major things to do in the configuration code are to create an Agent object that manages instrumentation, to register payload functions, and to initiate instrumentation.

4.3 Using Injector

The injector can be used to inject the agent shared library into a process. It can also be injected to monitor all the processes running on a specific port. The executable version of injector can be found inside $\$SP\_DIR/\$PLATFORM$. To instrument the function calls inside a particular process with process id *pid*, use the following command
$./injector pid *pid* LIB_NAME
Similarly to instrument all the processes running on a specific port *port*, use the following command
$./injector port *port* LIB_NAME

4.4 Extending SPI

4.4.1 Parser

4.4.2 Event

4.4.3 AddrSpace

4.4.4 CFG

4.4.5 Point

4.4.6 Instrumentation Workers

4.4.7 IPC Workers

5 Class Reference

5.1 Public Interface

5.1.1 Class Agent

Declared in: src/agent/agent.h

```
void SetInitEntry(string);


void SetInitExit(string);


void SetLibrariesToInstrument(const StringSet& libs);


void SetFuncsNotToInstrument(const StringSet& funcs);


void EnableParseOnly(const bool yes_or_no);
```

```
void EnableIpc(const bool yes_or_no);


void EnableHandleDlopen(const bool yes_or_no);


void EnableMultithread(const bool yes_or_no);


void Go();
```

5.1.2 Class SpPoint

Declared in: src/agent/patchapi/point.h

```
bool tailcall();


SpBlock* GetBlock() const;


SpObject* GetObject() const;
```

5.1.3 Class SpObject

Declared in: src/agent/patchapi/object.h

```
std::string name() const;
```

5.1.4 Class SpFunction

Declared in: src/agent/patchapi/cfg.h

```
SpObject* GetObject() const;


std::string GetMangledName();


std::string GetPrettyName();


std::string name();
```

5.1.5 Class SpBlock

Declared in: src/agent/patchapi/cfg.h

```
SpObject* GetObject() const;


in::Instruction::Ptr orig_call_insn() const;
```

5.1.6 Class SpEdge

Declared in: src/agent/patchapi/cfg.h

5.1.7 Utility Functions

Declared in: src/agent/payload.h

Utility functions are used when writing payload functions.

```
SpFunction* Callee(SpPoint* pt);
```

> Returns an instance of SpFunction for current function call. The argument *pt* represents the instrumentation point for current function call. If it fails, it returns NULL.

```
bool IsInstrumentable(SpPoint* pt);
```

```
void Propel(SpPoint* pt);
```

```
bool IsIpcWrite(SpPoint*);
bool IsIpcRead(SpPoint*);
```

```
struct ArgumentHandle;
void* PopArgument(SpPoint* pt,
                  ArgumentHandle* h,
                  size_t size);
```

5.2 Private Interface

5.2.1 Class SpParser

Declared in: src/agent/parser.h

```
SpObject* exe() const;
```

5.2.2 Class SpContext

Declared in: src/agent/context.h

5.2.3 Class SpEvent

Declared in: src/agent/event.h

5.2.4 Class SpPropeller

Declared in: src/agent/propeller.h

5.2.5 Class SpSnippet

Declared in: src/agent/snippet.h

5.2.6 Class SpAddrSpace

Declared in: src/agent/patchapi/addr_space.h

5.2.7 Class SpInstrumenter

Declared in: src/agent/patchapi/instrumenter.h

5.2.8 Inst Workers

Declared in: src/agent/inst_workers/

5.2.9 Class SpThreadMgr

Declared in: src/agent/thread_mgr.h

5.2.10 Class IpcMgr

Declared in: src/agent/ipc/ipc_mgr.h

5.2.11 Class SpChannel

Declared in: src/agent/ipc/channel.h

5.2.12 IPC Workers

Declared in: src/agent/ipc/ipc_workers/

A Installation

This appendix describes how to build self-propelled instrumentation from source code, which can be downloaded from `http://www.paradyn.org` or `http://www.dyninst.org`.

Before starting to build self-propelled instrumentation, you have to make sure that you have already installed and built Dyninst (refer Appendix D in Dyninst Programming Guide[1] for building Dyninst).

Building self-propelled instrumentation on Linux platforms is a very simple three step process that involves: unpacking the self-propelled instrumentation source, configuring and running the build.

A.1 Getting the source

Self-propelled instrumenation's source code is packaged in *tar.gz* format. If your self-propelled instrumentation source tarball is called *src_spi.tar.gz*, then you could extract it with the commands *gunzip src_spi.tar.gz ; tar -xvf src_spi.tar*. This will create a list of directories and files.

A.2 Configuration

After unpacking, the next thing is to set SP_DIR, DYNINST_ROOT, PLATFORM and DYNLINK variables in *config.mk*. DYNINST_ROOT should be set to path of the directory that contains subdirectories like dyninstAPI, parseAPI etc., i.e. within dyninst directory SP_DIR should be set to the the path of the current working directory (where self-propelled instrumentation is installed).

[1] `http://www.dyninst.org/sites/default/files/manuals/dyninst/dyninstProgGuide.pdf`

DYNLINK should be set *true* for building agent as a small shared library that relies on other shared libraries. Otherwise, set DYNLINK as *false* for buidling a single huge shared library that static-linked all libraries

PLATFORM should be set to one of the following values depending upon what operating system you are running on:

i386-unknown-linux 2.4	:	Linux 2.4/2.6 on an Intel x86 processor
x86_64-unknown-linux2.4	:	Linux 2.4/2.6 on an AMD-64/Intel x86-64 processor

Before building, you should also check whether LD_LIBRARY_PATH environment variable is set. If you are using bash shell, then open /.bashrc file and check if LD_LIBRARY_PATH is already present. If not, then LD_LIBRARY_PATH variable should be set in a way that it includes $DYNINST_ROOT/lib directories. If you are using C shell, then do the above mentioned tasks in /.cshrc file.

A.3 Building

Once config.mk is set, you are ready to build self-propelled instrumentation. Move to PLATFORM directory and execute the command *make*. This will build libagent.so, injector, test-cases and test-programs. Other make commands for custom building are as follows:

make spi	:	build injector and libagent.so
make injector_exe	:	build injector
make agent_lib	:	build libagent.so

That is it! Now, you are all set to use Self propelled intrumentation.

B Testing

C Directory Organization

- common.flag.mk : Compilation flags
- config.mk : User-defined configuration file
- spi.target.mk : Major compilation rules
- test.target.mk : Compilation rules for tests

- scripts/ : Containing some scripts to assist testing
- h/ : Public header files
- src/ : Source code for injector, agent, and tests
  - agent/ : Source code for agent
  - common/ : Common code used by injector and agent
  - injector/ : Source code for injector
  - test/ : Some tests
- user_agent/ : Example user-provided agents
- x86-unknown-linux-2.4/ : Build directory for x86-64
- i386-unknown-linux-2.4/ : Build directory for x86