Paradyn Parallel Performance Tools

Dyner Users's Guide

Release 4.0
May 2003

Jeffrey K. Hollingsworth & Mehmet Altinel
Computer Science Department
University of Maryland
College Park, MD 20742
Email: hollings@cs.umd.edu
Web: www.cs.umd.edu/projects/dyninstAPI

1. Introduction 3

2. Installation 3

2.1 INSTALLING TCL/TK 3
2.2 INSTALLATION AND COMPILATION OF DYNER 3

3. Dyner Commands 4

3.1 AT COMMAND 4
3.2 ATTACH COMMAND: 5
3.3 BREAKPOINT COMMANDS: 5
3.3.1 break command: *5*
3.3.2 listbreak command: *6*
3.3.3 deletebreak command: *6*
3.4 COUNT COMMAND: 6
3.5 DECLARE COMMAND: 6
3.6 DETACH COMMAND: 6
3.7 EXECUTE COMMAND: 7
3.8 KILL COMMAND: 7
3.9 LOAD COMMAND: 7
3.10 MUTATIONS COMMAND: 8
3.11 PRINT COMMAND: 8
3.12 REMOVE CALL COMMAND: 8
3.13 REPLACE COMMAND: 8
3.14 RUN COMMAND: 9
3.15 SAVE COMMAND: 9
3.16 SAVESTART COMMAND: 9
3.17 SHOW COMMAND: 9
3.18 SOURCE COMMAND: 9
3.19 TRACE COMMANDS: 9
3.20 WHATIS COMMAND: 10

4. Dyner Examples 10

4.1 OUTPUT REDIRECTION 11
4.2 CONDITIONAL BREAK POINTS: 11
4.3 MODIFICATION AND PERFORMANCE MEASURING 12
4.4 FUNCTION TRACE 12

References 12

1. INTRODUCTION

Dyner is a TCL-based interactive command line tool based on DyninstAPI library. DyninstAPI allows users to insert code into a running program and change or remove subroutine calls from application program. A wide variety of applications can benefit from these features including performance monitoring and debugging applications. Detailed description of DyninstAPI can be found in [1,2]. Dyner provides extra functionality on top of DyninstAPI with its command language such as declaration of new variables, navigational commands for type system, function trace facility.

2. INSTALLATION

2.1 Installing Tcl/Tk

The user interface of Dyner is based on the Tcl package, and therefore Tcl is required in order for it to run. Tcl (and Tk, which is not used by Dyner) can be downloaded at the URL: http://tcl.activestate.com/.

On Irix, Dyner is built as a 64-bit executable and requires a 64-bit version of Tcl. However, the configure script that comes with Tcl does not currently support building a 64-bit version for Irix. A modified, unsupported 64-bit binary version of Tcl for Irix can be downloaded at the Dyninst site, http://www.dyninst.org/ (click on the link for the current Dyninst release).

2.2 Installation and compilation of Dyner

The Dyner source is located in two directories: src and tests. The src directory contains the required files for Dyner, whereas the sources of the tests programs are provided in the tests directory. In the current version, the files in src directory are:

- dyner.C: This file contains all the dyner commands.
- cmdline.l and cmdline.y: These are lex/yacc files that are used to construct Dyner parser which generates DyninstAPI snippets from input Dyner statements.
- dynerList.h and breakpoint.h: Utility files.

For each platform that Dyner is available, there is a separate directory that includes corresponding makefiles to build the dyner executable.

In order to compile Dyner, the flex and bison utilities are needed. These tools create intermediate files that are required in the Dyner executable. To build the Dyner executable just go into the cor-

responding directory depending on the platform and type gmake (or make depending on the system configuration). As the makefiles employ the configuration parameters from DyninstAPI configuration files, DyninstAPI should be installed and properly configured in the system.

3. DYNER COMMANDS

Dyner program is started by typing "dyner" at the shell. The target programs are loaded into Dyner with "load" command. All the commands except load commands require a mutatee program loaded in to the dyner. As Dyner is based on DyninstAPI library, some commands might not be available on different platforms. The full set of the commands is available on Sparc Solaris platform.

When a program is running, typing control-C will stop the application and return execution to the dyner command. Using the run command again will resume execution.

The following commands are available in Dyner:

3.1 at command

The at command is used for inserting snippet codes into mutatee program. It has two forms of usage:

- `at <function> [entry|exit|preCall|postCall] <statements>`

  The snippet code inserted with this command is called either at the entry/exit of the input function, or before/after the call of the function. The syntax of a statement is described below.

- `at termination <statements>`

  This form of "at" command allows users to execute snippet code when the execution of mutatee terminates. This command is implemented using Exit callback feature of DyninstAPI.

Dyner snippet statements are written in C like syntax. Dyner parses the input statements and then generates DyninstAPI snippet code which is inserted into mutatee. Local and global variables, and functions declared in the mutatee can be used in the statements. Function parameters can be accessed either by their names (If symbol table provides that information) or using '$<param index>' format. For example '$0' and '$2' represent first and third parameters of the input function respectively. Moreover, the dyner variables declared in the session can be included in the statements. The start and end of the statements are indicated by "{" and "}" characters. These characters are also used to declare blocks. Current version of the Dyner allows the following types of statements:

- arithmetic expressions (Assignments, ‘+’, ‘-‘, ‘/’, ‘++’, ‘--‘ operators)

- if and if-else statements (Boolean conditions are declared using ‘==’, ‘!=’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘&&’, ‘||’ operators)

- function calls

- statement blocks (Declared between ‘{‘ and ‘}’ characters)

For example the following dyner command inserts a code segment that prints the value of second parameter if it is greater than 0. This code is invoked at the exit of foo function:

```
% at foo exit { if ($1>0) { printf("Second parameter is %d\n", $1); } }
```

3.2 attach command:

This command attaches dyner to a running program whose process id and path name is given as the arguments. A new dyner session is created as a result of this operation. The syntax of the command is:

```
attach <pid> <program>
```

3.3 Breakpoint commands:

One of the main usage of dyner is that it can be used as a debugger. For this purpose, Dyner provides three commands:

3.3.1 break command:

Two types of break command are available in Dyner:

- `break <function> [entry|exit|preCall|postCall] [<condition>]`

  Sets a break point at specified points of theinput function. These points can include entry/exit of the function or before/after call of the function. Break points can be conditional if the user provides a boolean condition which is defined using a C like syntax. Dyner generates a unique number for each break point. For example the following break command causes the execution to stop at the entry of foo function if the second parameter equals to 0:

  ```
  % break foo entry ( $1 == 0 )
  ```

  Note that, break conditions are declared between ‘(‘ and ‘)’ chracters.

- `break <file name:line number> [<condition>]`

Sets an arbitrary break point at specified line number of the input file. As in the previous form, a condition can be given for this type of breakpoints.

3.3.2 listbreak command:

This command simply displays the list of break points declared in the dyner session so far. This command does not take any arguments.

3.3.3 deletebreak command:

Users can delete the break points using this command by supplying the break point number as argument. The syntax is:

```
deletebreak <breakpoint number ...>
```

3.4 count command:

Dyner can also be used as a simple performance monitoring tool. The count command is provided for this purpose. It shows how many times the input function is called at the end of execution. Its syntax is:

```
count <function>
```

This command is actually implemented using other Dyner commands as shown below:

```
% declare int _<function>_cnt
% at main entry { _<function>_cnt=0; }
% at findSQ entry { _<function>_cnt++; }
% at main exit {printf("<function> is called %d times\n",_<function>_cnt);}
```

3.5 declare command:

This command allows users to create new dyner variables that are used in the modification of mutatee. Variables declared in this way can be used in the snippets as if they are declared in the mutatee. Format of declare command is:

```
declare <type> <variable>
```

Dyner variables play important role for monitoring and performance measuring.

3.6 detach command:

Sometimes it is desired to remove all the code inserted into mutatee. Detach command provides this functionality. It does not take any arguments.

3.7 execute command:

Dyner also allows users to execute snippets without inserting them into the mutatee. The execute command causes the specified snippet to be evaluated only once. Its syntax is:

```
execute <statements>
```

For example, in the following execute command 5 is assigned to a Dyner variable named var:

```
% execute { var = 5; }
```

3.8 find command:

Users can get type information about mutatee using this command. It can be invoked in four forms:

- find function <function name>

  This command will print all functions matching <function name> in the application image.

- find function <POSIX regex pattern>

  This command will print all functions matching <POSIX regex pattern> in the application image.

- find function <function name> in <module name>
- find function<POSIX regex pattern> in <module name>

  These commands function as the first two, but limit their search to the scope of the module specified by parameter <module_name>

3.9 kill command:

This command simply terminates the execution of the mutatee. It takes no arguments.

3.10 load command:

This command allows users to load either an executable, a shared library or a C++ source file. The usage of them are given below:

- `load <program> [arguments] [< filename] [> filename]`

A mutatee executable is loaded with this command. Each load of a mutatee creates a new Dyner session and all the break points, dyner variables and snippet codes from previous session are deleted before the new session begins.

- `load library <lib name>`

  This command loads a dynamically linked library into address space.

- `load source <C++ file name>`

  This command first creates a dynamically linked library from a C++ source file and then loads it to the address space. All the functions and variables in the source file will be available for instrumentation.

3.11 Mutations command:

This command enables or disables the execution of snippets in the mutatee. Its syntax is:

```
mutations [enable|disable]
```

3.12 Print command:

The data type and value of a dyner variable are displayed with this command. Its syntax is:

```
print <Variable>
```

3.13 remove Call command:

Users can delete all the calls or a specific call (i.e. nth call) in a function using this command. It has the following syntax:

```
removecall <function>[:n]
```

3.14 replace command:

This command replaces function calls with another function. It can be used in two form:

- `replace function <function1> with <function2>`

  This command replaces all calls to <function1> in mutatee with calls to <function2>.

- `replace call <function1>[:n] with <function2>`

  This command replaces all the calls or a specific call (i.e. nth call) in <function1> with the function <function2>.

3.15 run command:

This command starts the execution of mutatee or continues the execution if the execution is stopped in a break point. It does not take any arguments.

3.16 save command:

This command saves the mutated binary to a new executable file. Call saveStart before calling this command.

3.17 saveStart command:

This command tells dyner to start tracking instrumentation to be saved with the save command. Only instrumentation inserted after saveStart is called will be save in the mutated binary.

3.18 show command:

Users can get type information about mutatee using this command. It can be invoked in three forms:

- `show [modules|functions|variables]`

  This command displays module names, global functions and variables declared in the mutatee.

- `show functions in <module>`

  The list of functions declared in a module is provided with this command.

- `show [parameters|variables] in <function>`

  This command displays local variables or parameters declared in the input function.

3.19 source command:

To allow users to re-use frequently used commands over and over again, Dyner provides the source command which takes a file name as an argument and executes the dyner commands stored in that file (For those familiar with TCL, this is the standard source command). This command is invoked in the following form:

```
source <file name>
```

3.20 trace commands:

One of the main difficulties in instrumenting a mutatee is finding the right functions to instrument. In order to avoid instrumenting unnecessary functions, Dyner provides trace commands

that allow users to see which functions are actually called in the execution of the mutatee. Trace command can be used in two forms:

- `trace <function>`

  This command makes necessary modifications in the input function to print a message at the entry and exit of the function.

- `trace functions in <module>`

  The previous form of the trace command works on only one function. However, in many cases tracing all the functions in a module is desired. This command provides this functionality: it traces all the functions declared in the input module.

In order to reduce the number of functions traced in the mutatee, Dyner provides the untrace command. The untrace command removes the effects of trace command. As in the trace command, it can work on a single function or all the functions declared in a module. The forms of the command are:

```
untrace <function>
untrace functions in <module>
```

3.21 whatis command:

This command displays detailed information about variables and functions declared in the mutatee. Local variables and parameters are accessed when the function name is also provided with "-scope" option. Output information includes the type, scope, line number and frame offset of the variable. Its syntax is:

```
whatis [-scope <function>] <variable>
```

4. DYNER EXAMPLES

In this section we show several Dyner sessions to demonstrate the use of the commands. The following program will be used in the examples.

```
#include <stdio.h>

int findSQ(int inp) {
        int res = inp * inp;

        return res;
}

int AddTwo(int inp) {
        return inp+2;
}
```

```
int main() {
        int val = 2;

        val = findSQ(val);
        val = findSQ(val);
        val = findSQ(val);

        printf("Value %d\n", val);
        return 0;
}
```

4.1 Output Redirection

This example adds code to the mutatee so that all output that the mutatee writes to its standard output file descriptor is copied to a file named "dyner.out" (so it works like "tee," which passes output along to its own standard out while also saving it in a file). The motivation for the example session is that you run the mutatee, and it starts to print copious lines of output to your screen, and you wish to save that output in a file without having to re-run the program. Actual implementation of this program with using DyninstAPI is provided in DynistAPI users guide. Note that parameters of _write function are accessed by '$' character followed by parameter number.

```
% load testDyner
Loading "testDyner"
% declare int flagVar
% declare int fd
% at _write entry { if (flagVar==0) {fd=open("dyner.out", 257); flagVar=1;}
if ($0==1) write(fd, $1, $2); }
% run
Value 256

Application exited.
%
```

4.2 Conditional Break Points:

The following example shows how dynamic break points can be set in Dyner. In this example the execution of mutatee is stopped at the third call of function findSQ.

```
% load testDyner
Loading "testDyner"
% declare int cnt
% at findSQ entry { cnt++; }
% break findSQ entry ( cnt==2 )
Breakpoint 1 set.
% run

Stopped at break point 1.
 1: findSQ (entry), condition
% print cnt
    cnt = 2
% run
Value 256

Application exited.
```

```
%
```

4.3 Modification and Performance Measuring

This example shows how Dyner can be used to modify mutatee program and to get basic performance data. Modification is done by replacing the second function call in main with the AddTwo function. Note that findSQ is called only 2 times and output of the testDyner executable is changed to 36.

```
% load testDyner
Loading "testDyner"
% declare int cnt
% at main entry { cnt = 0; }
% at findSQ entry { cnt++; }
% at termination { printf("findSQ called %d times\n", cnt); }
% replace call main:2 with AddTwo
% run
Value 36
findSQ called 2 times

Application exited.
%
```

4.4 Function Trace

The following example shows how the trace command helps to understand which functions are called during the execution of mutatee.

```
% load testDyner
Loading "testDyner"
% replace call main:3 with AddTwo
% trace functions in testDyner.C
% run
Entering function main
Entering function findSQ
Exiting function findSQ
Entering function findSQ
Exiting function findSQ
Entering function AddTwo
Exiting function AddTwo
Value 18
Exiting function main

Application exited.
%
```

REFERENCES

1. B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications,*2000.
2. J. K. Hollingsworth and B. Buck, " DyninstAPI Programmer's Guide", http://www.dyninst.org/rel2.3/dyninstProgGuide.v23.pdf , March 2001