Experience Report on Using the dyninstAPI

Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik der Technischen Universität München
D-80290 München

1 Introduction

The following report summarizes my experience with the dyninstAPI during my research visit at Univ. of Wisconsin, Madison. My primary goals w.r.t. dyninstAPI have been to acquaint myself with the API and its usage, and to figure out, whether it can be used within the OMIS compliant monitoring system OCM, developed at LRR-TUM. The report is therefore divided into two main sections covering these topics.

The platforms for these experiments have been a PC running Solaris 2.6 and an Ultra-SPARC Server running Solaris 2.6[1]. I used release 1.2 of the dyninstAPI. So the experiences stated below only apply to these configurations.

2 Experience in Using the dyninstAPI

To acquaint myself with the dyninstAPI, I wrote a small tool that generates a procedure entry/exit trace of its mutatee (i.e. the monitored program), using the dyninstAPI test program `test1.C` as a basis. The tool achieves its task by dynamically linking a shared library to the mutatee process. This library contains a function `_fnprint` printing the trace entries. Then, dyninstAPI is used to instrument the entry point and exit points of *every* function with a call to `_fnprint`. See the source files and the `README` file for more details[2].

I was able to have a basic version of the tool operational after 2 or 3 days. Without an interface like dyninstAPI, it would have taken much longer to develop this tool. For this specific tool, an implementation using the `/proc` file system and traps could be done. Function addresses could be determined by calling `nm` or `elfdump`. Exit points of functions could be instrumented by placing a trap at the function's return address when the function is entered. The main problems for the implementation would be the handling of dynamically linked libraries and the extensive bookkeeping needed to manage the temporarily inserted breakpoints at return addresses and their mapping to the names of the returning functions.

However, I still needed more time to get the tool working than I think should have been necessary. My problem was not the understanding of the concepts of dynamic instrumentation, as I knew from several talks and papers, what dyninst is doing and how it works. Rather, there has been a couple of small problems concerning the documentation, the API, and the implementation. I will identify them in the following sections and try to suggest improvements.

2.1 Documentation

I refer to the *DyninstAPI Programmer's Guide* release 1.2, Sept. 23, 1998.

- The method `BPatch::formatErrorString()` is not documented. This gave me a hard time, since without that function, the error callback is not of much use. The manual should also provide an example on how to use `BPatch::getEnglishErrorString()` and this function in an error handler. This is currently not clear from the description in the manual.

[1] More specific: the machines have been `basil` and `e-fitz`.
[2] They are accessible at `~wismuell/public/FNTRACE`.

- The error numbers passed to the error callback should be documented, since a tool sometimes may want to handle certain errors differently.
- The documentation for `BPatch_thread::loadLibrary()` should provide more background information. First, it should indicate that the method can be used to include functions into the mutatee, which can be called by instrumentation code, thus allowing to perform even complex actions within the instrumentation. Second, it would be nice to have some documentation on how to create a shared library. Although this is not directly related to dyninst, I expect that a lot of poeple (like me) didn't create a shared library before. Last, the limitations of the current implementation of dyninst should be mentioned, especially the fact, that the library should only contain position independent code.

  While the first of these items was more or less clear to me, I first could not manage to correctly create the shared library, since I didn't generate position independent code.

2.2 Interface

At the beginning, I was confused by the fact, that sometimes the interface works with pointers to objects, while at other times it works with the objects themselves, e.g.:

- `BPatch_thread::malloc()` returns a pointer to an object, whereas `BPatch_thread::free()` requires the object itself as its parameter. This is especially confusing, since it it different from what you are used to from the UNIX `malloc/free` routines.
- `BPatch_image::findVariable()` returns a pointer to a `BPatch_variableExpr`, whereas e.g. `BPatch_paramExpr()` returns the object itself.

Of course, I am aware that efficiency reasons and other implementation issues in C++ are the reason for these inconsistencies. However, they make the use of the API more difficult than necessary.

The same tradeoff between efficiency and ease of use (and security) also applies to the template class `BPatchVector<BPatch_snippet *>`. In my opinion, it would be better to have a class `BPatchVector<BPatch_snippet>` instead, if the `BPatch_snippet` objects are not extremely large[3]. In `fntrace`, I had to program a loop iterating over all parameters of the instrumented function. It would have been much easier and clearer to write something like:

```
for (i ...) {
    BPatch_paramExpr expr(i);
    params->push_back(expr);
}
```

instead of having to use `new` and `delete` to dynamically allocate and destroy the `BPatch_paramExpr`'s. There are also some security issues here:

- The documentation doesn't state when it is safe to delete these objects again.
- Looking at the example codes, one is tempted to write something like

  ```
  for (i ...) {
      BPatch_paramExpr expr(i);
      params->push_back(&expr);
  }
  ```

  which does not work, as the compiler will destroy `expr` at the end of the loop.

[3] The problem, of course, is that they can get arbitrarily large, as they basically are syntax trees.

2.3 Error Reporting

In the current version of the dyninstAPI, error reporting appeared as one of the weak points to me. As already mentioned, the function actually returning the error messages is not specified in the programmer's manual. Furthermore, the example in the manual does not include error handling at all. So at the beginning of my experiments, I just got a NULL pointer as the result of `createProcess`, but I didn't have any idea what went wrong. With a properly installed error callback, the situation gets better (e.g. it tells me that I should set the `DYNINSTAPI_RT_LIB` environment variable). However, I experienced a lot of situations where I just got the messages

```
Unable to open executable file: foo
Unable to parse image: foo
```

without more detailed reasons. The error message should at least contain a reason obtained by inspecting `errno` after the `exec` system call. I had a similar experience with the `BPatch_thread::loadLibrary()` method, which doesn't provide any error message at all when it fails. It is very difficult then to find out whether

- I didn't set `LD_LIBRARY_PATH` (the documentation should probably mention that)
- The library is not existent/accessible
- The library contains unresolvable symbols (this is especially hard to figure out, since I didn't think of this problem at all)

Getting more detailed error messages may not be easy, since this method is implemented via an inferior RPC, so the actual error status is in the mutatee process. However, they are really important for usability.

2.4 Generating dyninstAPI

In my later experiments to combine dyninstAPI with the OCM, I had to make some modifications to dyninst (see below) and recompile it. I noticed two problems, which should be fixed, although they are not really serious.

- The `make`-run does not stop when a compilation error is found. It just aborts the processing of the current subsystem and continues with the next one. The problem is that at the end of the `make`-run, there is no clear indication that an error has occurred. So you always have to scan the whole output for error messages.
- The assembler needs an incredible amount of virtual memory while processing the file `template1.C`. So probably this file should be split in two (or more) files. I actually had to close a couple of applications on `basil` (which has a reasonable amount of 175 MB swap space) to compile this file.

2.5 Bugs

Since `fntrace` instruments *every* function it can find, it turned out to be very good in finding cases where the instrumentation done by dyninst is incorrect. So at first I basically got core-dumps instead of function traces. I tracked down (to a different extent) all the problems I encountered when applying `fntrace` to simple programs like `ls` or `env`. The problems are documented in the source code (`fntrace.C`) and the `PROBLEMS` file in the same directory.

Hopefully, this helps to make dyninst more stable. `fntrace` could also serve as an additional test program for the API.

3 Interoperability Between dyninstAPI and Other Tools

3.1 Combining OCM and dyninst

After I gained experience with using dyninst for a stand-alone tool, I started to explore the possible combination with our OCM monitoring system. I believe that my situation here was very similar to that of any other tool (or more specific: debugger) developer who thinks about using the dyninstAPI: The OCM already has its own code to create processes, and it uses the `/proc` and/or `ptrace` interfaces to control and monitor these processes. So in the first step, I did not want to completely change this code, such that it exclusively relies on the dyninstAPI functions[4].

Rather, the idea was to attach dyninst to an existing process, when the first request is executed that uses dyninst. This means that both the OCM and dyninst are attached to the process at the same time. So the question was whether this would work.

To provide more detail: my goal was to implement a monitor extension for the OCM that offers two new services:

- An event service that monitors the entry into arbitrary functions and also allows access to the function's parameters. This event is thus very similar to the `thread_has_started_lib_call` event specified in the OMIS document, but it is not restricted to a predefined set of functions.
- A manipulation service that offers an inferior RPC, i.e. allows to call any function in the context of the application process.

From my experiments with `fntrace` I already knew how to implement the necessary instrumentation. So the interesting questions that remained were:

- Can I attach dyninst to an existing process that is already controlled via the `/proc` interface? There are two problems here:
  - OCM and dyninst may require conflicting settings of the various flags and options provided by `/procs` `ioctl` interface.
  - The process may be in a state that is not expected by dyninst.
- Can I execute an inferior RPC using `BPatch_thread::oneTimeCode` while the process is controlled by the OCM via `/proc`?
  Again, the process may be in any state when I request the inferior RPC.
- Can I use dyninst to dynamically load the OCM library to the monitored process?
  Since my goal was to monitor function calls in application processes without requiring special preparation, this means that the OCM code needed for action execution and communication with the monitor process must be linked to these processes at runtime.

3.2 Problems and Experiences with OCM/dyninst

It turned out that dyninst did a surprisingly good job, taking into account that (at least to my knowledge) this was the first attempt ever to combine dyninst with a debugger-like tool. There were, of course, some problems, but they should mostly be accounted to the OCM rather than dyninst. I give a brief description of these problems in the next paragraphs.

I expected to get the most severe problems when attaching dyninst to a process that has already been created and is monitored by the OCM, as this includes initialization of dyninst, loading libraries etc. In order to avoid bigger problems in the beginning, I decided to detach the OCM from the process before attaching dyninst, and re-attaching again later. My idea was that except for the process being stopped, the situation should then be equal to the 'normal' one where dyninst attaches to a running process.

In the first approach, the process would just hang, after I tried to attach dyninst. The reason for this behavior was the fact that the OCM did not reset the bits instructing `/proc` to monitor system calls and signals. On the other hand, dyninst assumed that these bits are cleared when it attached to the process, and thus did not handle the notifications properly. It's not fully clear who is to blame here, the OCM or dyninst. In principle, both are to blame: the OCM failed to restore the previous state of `/proc`, while dyninst relied on its default settings. Since I knew the OCM code better than the dyninst code, I fixed the problem in the OCM. It turned out to be not trivial to really clean up everything when detaching from the process, since the documentation of `/proc` does not document its internal state components too well. So I needed several iterations to get things right.

[4]At the time of this writing, I'm not even sure whether the API is general enough to allow this.

Once I could attach dyninst to the monitored process, dynamically linking the OCM library to that process and initializing it turned out to be very easy. The main reason for this was the fact that all functions and other data in the OCM library are accessible via a single symbol in this library. Thus, I only needed to determine the address of this symbol in the dynamically linked library to do the initialization.

The next real problem occurred after I re-attached the OCM. When the monitored process hit a breakpoint set by the OCM, the OCM suddenly did not recognize the breakpoint anymore (on x86 Solaris). The reason was that `/proc` offers a flag (`PR_BPTADJ`) to adjust the PC after a trap has been executed, so the PC points to the trap rather than the following instruction. dyninst set this flag, whereas the OCM relied on this flag not being set. This was the only case, where dyninst and OCM really had conflicting needs. Fortunately, it turned out that dyninst already contained code allowing for the case where the PC points to the instruction after the trap, so it really didn't need to set this flag. Since I didn't want to modify dyninst, I explicitly reset this flag in the OCM's attach code. However, it may be worth thinking about removing the code to set this flag from dyninst's initialization code. From the viewpoint of supporting interoperability, having this flag in `/proc` seems to be a bad idea.

It should be noted, that what I did with the OCM and dyninst is more or less the best case situation for interoperable tools, since the tools don't really work at the same time. I.e. the monitored process is either controlled and/or modified by OCM or by dyninst, but not by both at the same time. So there is little conflict between the two.

3.3 Combining dyninst and `gdb`

I tried to achieve a similar kind of interoperability when implementing the `fntrace` program. To comfortably track down the instrumentation problems in dyninst, I tried to implement an option in `fntrace` that results in the following behavior:

- `fntrace` creates and instruments the mutatee using dyninst
- after instrumentation, the mutatee is not continued, but instead dyninst is detached
- `gdb` is started on the mutatee

My goal was that I could then use `gdb` to inspect the instrumentation, to set breakpoints and to continue the mutatee.

It turned out that on x86 Solaris, the mutatee just dumped core when I continued it with `gdb` (or by using `prun`). One reason was that dyninst uses an illegal instruction to stop the mutatee after it has executed the initialization routine in the dyninst run-time library. When detaching, dyninst didn't clear the resulting `SIGILL` signal, so it was forwarded to the mutatee when `gdb` or `prun` continued it. Again, this is an example of insufficient clean-up. However, the really interesting observations have been:

- When I integrated OCM and dyninst, the same situation occurred, but here it worked. The reason why it worked was actually an interoperability problem in the OCM: Whenever the OCM detects that the monitored process has stopped, it stores the reason causing the stop in its internal process data structure. When continuing the process, the OCM decides whether or not to forward the signal to the process, based on the contents of this date structure. Since the process was not controlled by OCM when it was stopped by the `SIGILL`, the OCM was just unaware of the pending signal and thus did not forward it to the process. This behavior is of course incorrect in the general case.
- The same situation on SPARC Solaris did work. The only difference is that on SPARC, dyninst uses a trap to stop the mutatee. Thus there is a pending `SIGTRAP` signal. Obviously, gdb does never forward this signal to the debugged process.
- Even after I added code to dyninst to clear the pending signal, the program dumped core when I continued it in `gdb`. Things now worked well, however, when I used `prun` to continue the process, proving that the cleanup was correct. `gdb` simply ignores the information stored in `/proc` about whether or not the signal should be forwarded, and decides on its own. It is, however, possible to instruct `gdb` not to forward `SIGILL` signals to the debugged process (`handle SIGILL nopass`). When doing this, using a version of dyninst that does not set the `PR_BPTADJ` flag, and instructing `fntrace` not to perform any instrumentation that uses traps, `gdb` finally works.

Instrumenataion using traps works with `gdb`, if it is instructed to forward `SIGTRAP` to the debugged process. However, `gdb` now also forwards the `SIGTRAP` generated by its own breakpoints, which causes the dyninst run time library to abort the process due to a failed assertion. What happens is that the signal handler installed by dyninst checks whether the signal was caused by dyninst's instrumentation, and aborts if this is not the case. The OCM seems to be smarter here. It forwards the signal to the debugged process, if and only if it was not caused by a breakpoint inserted by OCM, thus not causing this problem.

3.4 Summary

To summarize my experiences: I currently looked at the case, where two run-time tools are used on the same process one after another. For this case, the main lessons I have learned are:

1. Don't rely on default settings and/or 'usual' state when attaching
   Settings and state may have been changed by other tools.
2. Carefully restore previous settings and/or state when detaching
   Be nice to other tools that don't care about rule 1.
3. Be aware of other tools

3.5 Experience and Suggestions Regarding dyninst

An extremely helpful property of dyninst was the fact that after instrumenting, the mutatee can execute without a need for interacting with the mutator parts of dyninst. I.e. there is no need for the mutator process to periodically call the dyninst methods `BPatch::pollForStatusChange()` or `BPatch::waitForStatusChange()`. Calling these routines would not have been possible in the OCM, as the OCM relies on its own main loop, which waits for several different events at the same time. So we would have been forced to use a second process calling these functions, leading to a more complex scenario of interoperable tools, i.e. two tools controlling the same process at the same time.

Two long-term wishes regarding dyninst emerged from my experiments:

- It should be possible to attach dyninst to a process at any point in its execution. Currently dyninst assumes that the process already entered `main` when attaching. In the OCM scenario, this is usually not the case. When the OCM creates a process, it is stopped immediately before it executes its first instruction. Currently, dyninst cannot attach to the process at this point[5]. Of course, it would be possible to modify the OCM, so that it lets the program execute until it enters `main`, but this would make process creation in the OCM much more complex. In addition, C++ programs will execute code for constructors of static objects even before entering `main`.
- Another wish concerns the ability to access the parameters of instrumented functions. Currently, dyninst only allows to access them at the function's entry point. However, many functions return results via reference parameters or pointers. Usually, a tool wants to access these results at the function's exit point. Currently, this can only be done by saving the pointers at the function's entry point in a variable allocated using dyninst. The problem is, however, that dyninst only allows to allocate heap variables. This means that
  - there may be a huge overhead in space, as these variables must be permanently allocated for every instrumented function, even while the function is inactive.
  - recursive functions cannot be instrumented correctly.

  An alternative to providing access to parameters at a function's exit point could be to provide some means to allocate space on the stack.

[5] It is aborted by a failed assertion, since the process does not yet have a stack, i.e. the frame pointer is zero.