Dyninst: An API for Runtime Code Patching

presented by
Skylar Byrd Rampersaud
http://www.cs.umd.edu/~byrd/dyninst.ppt

The First Slide

• Goal: change a program while it is executing
  – Without recompiling, relinking or restarting
• Applications
  – Dynamic performance measurement
  – Performance steering in large-scale simulations
Process Model

• A program can attach to a running program
• Create a new bit of code
• Insert it into the program
• Can augment or change subroutines

Dyninst is Not

• An instrumenting compiler
• Adding binary code to an executable before it is run
• Machine code (assembly language)
Terminology

- Point - a location where code can be inserted
- Snippet – representation of executable code to be inserted
- Thread – thread of execution
- Image – the static on-disk program

Abstractions

Fig. 1 Abstractions used in the API
3 Main Interface Components

- Classes to manipulate executing code
  - BPatch, BPatch_thread
- Classes to access the original image and data structures
  - BPatch_module, BPatch_function
- Classes to construct and insert new code snippets
  - BPatch_point, BPatch_snippet

Statements to be Added

- A collection of BPatch_snippet instances (and subclasses representing specific types of code)
  - Collection forms a direct acyclic graph
  - Abstract Syntax Tree created from leaf to root
Types

• The API includes a simple type system
  – Integers, strings, floats
  – Support for aggregate types

Events

• API provides notification of application events
• Also provides a way to query for specific events
How Does It Work?

• Mutator process uses debugger-style OS functions to access memory and events of running process
• Translate snippets into machine code
• Copy code into an array in the running process
• Uses “trampolines” to for transferring execution to inserted code

Trampolines

• Replace some instructions with a branch to a base trampoline
• Base trampoline branches to a mini-trampoline
• Base trampoline executes the original instructions once execution returns from the mini
Mini-trampoline

- Saves registers and other state
- Contains code for one snippet
- Can chain these together to include multiple snippets at one point
- Branches back to the base trampoline at the end of the final snippet

Trampolines Illustrated

Fig. 2 Inserting code into a running program
Three Example Programs

- Procedure call counting
- RETEE
- Conditional breakpoints

Procedure Call Counting

- Mutator creates an instance of the BPatch class
- Identifies process (running or not)
  - Creates new thread or new process
- Defines snippets and points
  - Instrumenting a single function may require multiple points
- Creates a new variable in the target space
Counting Procedure Calls

```
1 auto batch;
2 Batch_thread *appThread = appThread->getThread(guiState, argV);
3 Batch_image *appImage = appThread->getImage();
4 Batch_Vector *batch_point = points -
5 appImage->findProcedurePoint("Interestingprocedure", Batch_entry);
6 Batch_variableExpr *intCounter =
7 appThread->malloc(appImage->findType("int"));
8 Batch_arithExpr addOne(Batch_assign, *intCounter,
9 Batch_mathExp(Batch_plus, *intCounter, Batch_constExpr(111)));
10 appThread->inserBlock(addOne, *points);
```

Fig. 5 Code to count the number of occurrences of "InterestingProcedure"

---

**RETEE**

- Uses the one-time code feature of the API

```
1 auto function *openImage = appImage->findFunction("open");
2 Batch_Thread Batch_threader = openImage;
3 Batch_constExpr fileImage(argV[0],)
4 openImage.push_back(fileImage);
5 Batch_constExpr fileImage[0] = openImage;
6 openImage.push_back(fileImage[0],)
7 openImage.push_back(fileImage[0],)
8 openImage.push_back(fileImage[0],)
9 Batch_funcCallExpr openCall(openImage, openImage)
10 Batch_variableExpr *fVar =
11 appThread->malloc(appImage->findType("int"));
12 Batch_arithExpr openExpbr(Batch_assign, *fVar, openCall);
13 appThread->osw7FreeCode(openExpbr);
```

Fig. 6 Code to open the log file in the application
Conditional Breakpoints

• Very slow in a traditional debugger
• Results averaged over 20 runs of the program

<table>
<thead>
<tr>
<th>Application</th>
<th>Breakpoints</th>
<th>Dyninst</th>
<th>gdb</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress56</td>
<td>32,513</td>
<td>0.06</td>
<td>74.35</td>
</tr>
<tr>
<td>li (simulator)</td>
<td>119,009</td>
<td>2.53</td>
<td>201.04</td>
</tr>
<tr>
<td>li (compare)</td>
<td>4475</td>
<td>6.90</td>
<td>18.98</td>
</tr>
<tr>
<td>li (binary)</td>
<td>401</td>
<td>29.69</td>
<td>21.62</td>
</tr>
</tbody>
</table>

Other Applications

• Online critical path analysis in SMPs
• Harmony
  – Use runtime observations to automatically tune programs
• Eliminate redundant synchronization in parallel programs
• Other debugging and performance monitoring tools
Related Work

• Binary editing tools
• ‘C
  – Allows a program to define a set of C-like statements and call them
• Instrumenting compilers
• Los Alamos Debugger

Conclusion

• Dyninst is a simple runtime API to allow creation and patching of programs
• Ability to create portable tools by providing machine-independent abstractions
• Implemented Platforms
  – Intel x86, Sun Sparc, Compaq Alpha, MIPS, IBM Power
• http://www.dyninst.org/