Runtime Program Evolution

Jeff Hollingsworth
Motivation

● Software systems are
  - becoming more complex
  - being built from component parts
  - running in complex and varied environments

● Tools are required to
  - understand the behavior of such systems
  - react to changing environments
  - manage software components
dyninstAPI

- API for runtime code patching
  - new code can be added to a program while it executes
  - permits instrumentation and modification of programs
- Provides processor independent abstractions
  - same patching can be applied to multiple systems
- Includes meta-instrumentation
  - tracks overhead on inserted code
Applications of Runtime Code Patching

- **Performance measurement**
  - Recording application behavior

- **Correctness debugging**
  - Fast conditional breakpoints
  - Data breakpoints

- **Execution driven simulation**
  - Architecture studies

- **Testing**
  - Code coverage testing
  - Forcing hard to execute paths to be taken
Advantages of Runtime Code Patching

- **No forethought needed**
  - No user inserted probes
  - No special compiling or linking
  - Start anytime during execution

- **Only insert code when needed**
  - No wasted checks for “disabled” code
  - Can add new code during execution
Structure of the Dyninst Library

**Mutator**
- Mutator App
- API
- Dyninst Code
- Machine Dependent Code
- Ptrace or procfs

**Mutatee**
- Application Code
- Snippets
- Run-time Library
API Library

• Provides:
  - Functions for control of mutatee
  - Runtime code generation
  - Information about mutatee

• A set of C++ classes
  - Bpatch_thread
  - BPatch_image
  - BPatch_snippet
  - BPatch_variableExpr
  - BPatch_block
Representing Code Snippets

- **Platform Independent Representation**
  - Same code can be inserted into apps on any system

- **Simple Abstract Syntax Tree**
  - Can refer to application state (variables & params)
  - Includes simple looping construct
  - Permits calls to application subroutines

- **Type Checking**
  - Ensures that snippets are type compatible
  - Based on structural equivalence
  - Allows flexibility when adding new code
Snippet Example

if (flagVar == 0) fdVar = open(filename, ...)

BPatch_ifExpr
  BPatch_boolExpr(BPatch_eq, ...)  BPatch_arithExpr(BPatch_assign, ...)
    BPatch_variableExpr flagVar    BPatch_constExpr(0)
        BPatch_funcCallExpr
          BPatch_function "open"
            BPatch_Vector
              BPatch_constExpr(filename)
              BPatch_constExpr(O_WRONLY | O_CREAT)
              BPatch_constExpr(0666)
Type Support in Dyninst

- Access to local (stack) variables
- Complex types
  - non-integer scalars
  - structures
  - arrays
- Correctness debugging
  - print contents of data structures
Implementation

- **Use Compiler debugger info (stab records)**
  - access to user defined types
  - information about local variables
  - type information for all variables
  - line number to text segment address mapping

- **Incremental parsing**
  - parse stabs for a module on first use

- **dyninst User can define types**
  - allows the creation of new types for patched code
  - permits reconstruction of stripped symbols
API Example

// find all variables defined in an image
BPatch_Vector<BPatch_variableExpr *> vars = appImage->getGlobalVariables()

for (i=0; i < vars->size(); i++) {
    BPatch_variableExpr *v = (*vars)[i];
    switch (v->getType()->type()) {
    case BPatch_scalar:
        printf("%s is a scalar of type %s\n", v->getName(), v->getType()->getName());
        break;
    case BPatch_structure:
        FieldVector *fields = v->getType()->getComponents();
        for (j=0; j < fields->size(); j++) {
            Bpatch_field *f = (*fields)[j];
            printf("field %s is of type %s\n", f->getName(), f->getType()->getName());
        }
        break;
    }
}
Code Coverage Testing Using Dyninst

- **Code Coverage**
  - identifies source code lines not executed
  - ensures each basic block is taken at least once

- **Using Dyninst**
  - Allows use on arbitrary binaries
  - Permits removing code once a block is covered
    - Long running programs can be tested faster
  - Permits incremental instrumentation
    - First instrument function entry
    - On first call, instrument function’s blocks
Using Dominators to Reduce Counters

CFG

Dominator Tree

- instrument basic blocks that are leaf nodes in dominator tree
- Also instrument basic blocks with outgoing edge(s) to blocks not dominated by them
Postgres with Wisconsin benchmark

coverage for postgres

execution time for postgres

percentage

time(interval)

execution time (sec)

original 0 2 10 25 deletion interval (sec)

all basic blocks

dominator info

University of Maryland
Slow down

slow down ratio wrt. original execution

- purecov
- all basic blocks
- dominator info

tested programs

postgres  vortex  perl  mk88sim  jpeg

University of Maryland
Dyner Command Utility

- **TCL-based command line tool**
  - provides access to most dyninst features
  - easier to program for simple applications
  - can be used as a simple command-line debugger
    - fast conditional breakpoints
    - dynamic addition of printfs

- **Command Summary**
  - declare: create a new variable in the application
  - cbreak: insert conditional breakpoint
  - print: show contents of application data structures
  - at: insert a code snippet into the application
  - load, run, exit: process creation and manipulation
TCL Command Example

% load application
% declare int counter
% at main entry { counter = 0; }
% at importantFunc entry { counter++; }
% at main exit {
    printf("function called %d times\n", counter);
}
% run
Dyninst Status

- **Supported platforms**
  - SPARC (Solaris)
  - x86 (Solaris, Linux, NT)
  - Alpha (Tru64 UNIX)
  - MIPS (IRIX)
  - Power/PowerPC (AIX)

- **Software available on the web**
  - Includes TCL command tool (soon)
Expanding the Application/System Interface

Past Model:

Start program execution, hope for best

New Model:

Application exposes alternatives
  different algorithms/parameters
  performance expectations for options
System adapts application to optimize execution
Harmony Structure

- System
- Controller
- Applications and Libraries
- Metric Interface

Tuning Control
- Tuning Control
- Tuning Options
- Resource Requirements

Capacities and Requirements
- Capacities and Requirements

Availabilities and Requirements
Features of Harmony RSL

- **Bundles**
  - primary unit of adaptation
  - mutually exclusive sets of application options

- **Resource Requirements**
  - expected utilization for each option and resource

- **Performance Prediction**
  - expected performance of selected bundles
  - allows optimizing multiple applications on a system
Bundles

- **node**
  - CPU speed/disk capacity/available memory

- **link**
  - latency/bandwidth/protocol between nodes

- **communication**
  - entire application’s communication requirements

- **performance**
  - entire application’s performance

- **granularity**
  - switching between options at runtime
Harmony API

harmony_startup(<unique id>, <use interrupts>)

harmony_bundle_setup(“<bundle definition>”)

void *harmony_add_variable(“name”, <default>, <type>, <func>)

harmony_wait_for_update()

harmony_end()

Used by application to:
- define options
- learn of harmony selections
- receive information about the environment
Example: Client-Server Database

Query-Shipping

Client

Server

Data-Shipping

Client

Server
harmonyBundle Dbclient:1 where {
    {QS  
        {node  server
        {seconds 9}
        {memory 20}}
        
        {node  client
        {seconds 1}
        {memory 42}}

        {link client server 2}
    }

    {DS  
        {node  server
        {seconds 1}
        {memory 20}}
        
        {node  client
        {memory >=17}
        {seconds 9}}

        {link client server {44 + (client.memory > 24 ? 24 : client.memory) - 17}}
    }
}
Clients added one at a time:
- First two clients run with query-shipping
- Third client flips all to data-shipping
Results from PSTSWM

- Solves nonlinear shallow water equations
- Contains many options:
  - Multiple algorithms embedded in the code
  - Problem-specific options
  - Communication Parameters

<table>
<thead>
<tr>
<th>Size</th>
<th>Nodes</th>
<th>Min</th>
<th>Max</th>
<th></th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>T42L16</td>
<td>4</td>
<td>0.75</td>
<td>1.52</td>
<td></td>
<td>0.75</td>
<td>1.49</td>
</tr>
<tr>
<td>T42L16</td>
<td>8</td>
<td>0.50</td>
<td>1.03</td>
<td></td>
<td>0.50</td>
<td>0.77</td>
</tr>
<tr>
<td>T85L32</td>
<td>4</td>
<td>9.55</td>
<td>20.89</td>
<td></td>
<td>9.55</td>
<td>15.38</td>
</tr>
<tr>
<td>T85L32</td>
<td>8</td>
<td>5.99</td>
<td>11.41</td>
<td></td>
<td>5.99</td>
<td>7.90</td>
</tr>
</tbody>
</table>
Current Work

- **Application resource usage**
  - *potential*, not necessarily achievable
  - user, compiler, profiling

- **Performance prediction**
  - structural models
  - *POEMS*, AppLeS

- **Scheduling!**
  - Heuristics

- **More applications**
  - real-time vision, web server, video server
Active Harmony Conclusions

- **Launch and forget is not sufficient:**
  - Capacities are dynamic
  - Demands are dynamic

- **System-directed adaptation gives us:**
  - Complete information
  - Handles to running applications

- **But requires:**
  - Application restructuring (or layering, i.e. DSM)
  - Detailed resource requirements
Acknowledgements

- **Co-PIs**
  - Pete Keleher (Harmony)
  - Bart Miller (dyninst)

- **Graduate Students**
  - Harmony - Heonsang Eom, Dejan Perkovic, Cristian Tapus
  - dyninst - Bryan Buck, Mustafa Tikir

- **Research Staff**
  - Mehmet Altinel

- **Funding Agencies**
  - DARPA, DOE, DOD, NSF, NIST