# New APIs from P/D Separation

James Waskiewicz

Dyninst

# "Separation" completed

- Paradynd now uses the Dyninst API
  - Formerly made calls to the low-level code hidden by Dyninst
    - A development/testing nightmare
  - Now just links to libdyninstAPI
    - like any other mutator
  - End of a long, several-year process
- Brute-force final push:
  - Modify paradynd to use existing APIs as much as possible
  - Add new APIs to Dyninst as necessary
    - Functionality needed by Paradyn that was not previously available

# "Active" Snippet Insertion

- All instrumentation is now sanity-checked vs. current process state
  - Requires doing full stack walk(s) for each insertion
    - Stack walks are cached to improve performance in case of multiple insertions
  - Makes sure that snippets are not added to points that are currently executing inside instrumentation
    - Would cause re-writing of currently executing code (segfault)
- Insertion may change process state
  - Changes stackwalks for specific circumstances
    - Eg. Active call site (on the stack),
      - Modify stack frame to jump into instrumentation upon return.

# "Catchup" Snippet Execution Analysis

- Problem:
  - Atomic insertion of multiple snippets may imply a required sequence of execution
    - Might be violated, depending on where the program is stopped
  - Simple Example: (should do this in a diagram)
    - Snip1: At entry of foo(), turn on timer t
    - Snip2: At exit of foo(), turn off timer t
    - The program is stopped at point P, just after the entry point of foo()
    - User inserts Snip1 and Snip2 in an atomic operation at P and continues execution
    - Snip2 is executed, without Snip1 having preceeded it

# “Catchup” Analysis, con’t…

- Solution:
  - We cannot predict the intent of user snippets
  - But we CAN provide notification when any snippets in an insertion set fall after the current PC
- Requires full stack examination
  - For each thread
    - Much like we need to do for “active” insertions
- Q: Necessity or Value-add?
  - Most of the analysis for catchup is available by other means in Dyninst
    - Stack walks, address comparisons

# Added APIs

- Bpatch_process
  - Bool wasRunningWhenAttached()
  - Bool isMultithreadCapable()
  - Bool finalizeInsertionSetWithCatchup(…)
  - Bool oneTimeCodeAsync(…)  (overload)
- Bpatch_snippetHandle
  - getProcess()
- Bpatch_snippet
  - getCostAtPoint(Bpatch_point *p)

# Dyninst Object Serialization/Deserialization

## Binary for performance, XML for interoperability

# Why Binary Serialization (Caching)?

- Large Binaries
  - We've had reports of existing Dyninst analyses taking a prohibitively long time for large binaries (100s of MB)
    - Eg. Full CFG analysis of large statically linked scientific simulators
- More complex analyses are in the works
  - Dyninst continues to offer newer and more expensive-to-compute features
    - Control Flow Graphs
    - Data Slicing
    - Stripped binary analysis
  - Complex tools that use these analyses may find them cost-prohibitive
    - If they have to be re-performed every time the tool is run
    - Why not just save them?

# Caching policy

- Binary serialization should happen transparently
  - User-controlled on/off switch
    - Bpatch_setCaching(bool)
  - Granularity:
    - One binary cache file per library / executable
  - Checksum-based cache invalidation
    - Rebuild cache for a given binary when the binary changes
  - Example: libc is large and expensive to fully analyze, but it seldom changes
- Needs to support incremental analysis
  - User calls to API functions trigger on-demand analyses
  - Thus caching must also support incremental additions
    - Eg. Successive, more refined tool runs

# Why XML Serialization?

- Create standardized representations for
  - Basic symbol table information
  - Abstract program objects
    - Functions, loops, blocks….
  - More complex binary analyses
    - CFG, Data Slicing, etc…
- Exports Dyninst's expertise for easy use by
  - Other tools
  - Interfacing the textual world
    - Parse-able snapshots of programs
  - Cross-platform aggregation of results
- Allows Dyninst to use output from other tools in its own analyses
  - Other tools may perform different and/or richer analysis that would be valuable for Dyninst

# Unified serialization...

- Multiple types of serialization can share the same infrastructure
  - Leverage c++ and the Dyninst class hierarchy
  - Keep serialization/deserialization process as extensible as possible
    - Add new types of output down the road?
- Desired behavior:
  - serialize(filename, HierarchyRootNode, Translator);
    - Serialize hierarchy into <filename>
    - Traverse hierarchy in a (somewhat) generic manner
    - Translator uses overloaded virtual translation functions that can be specialized as needed

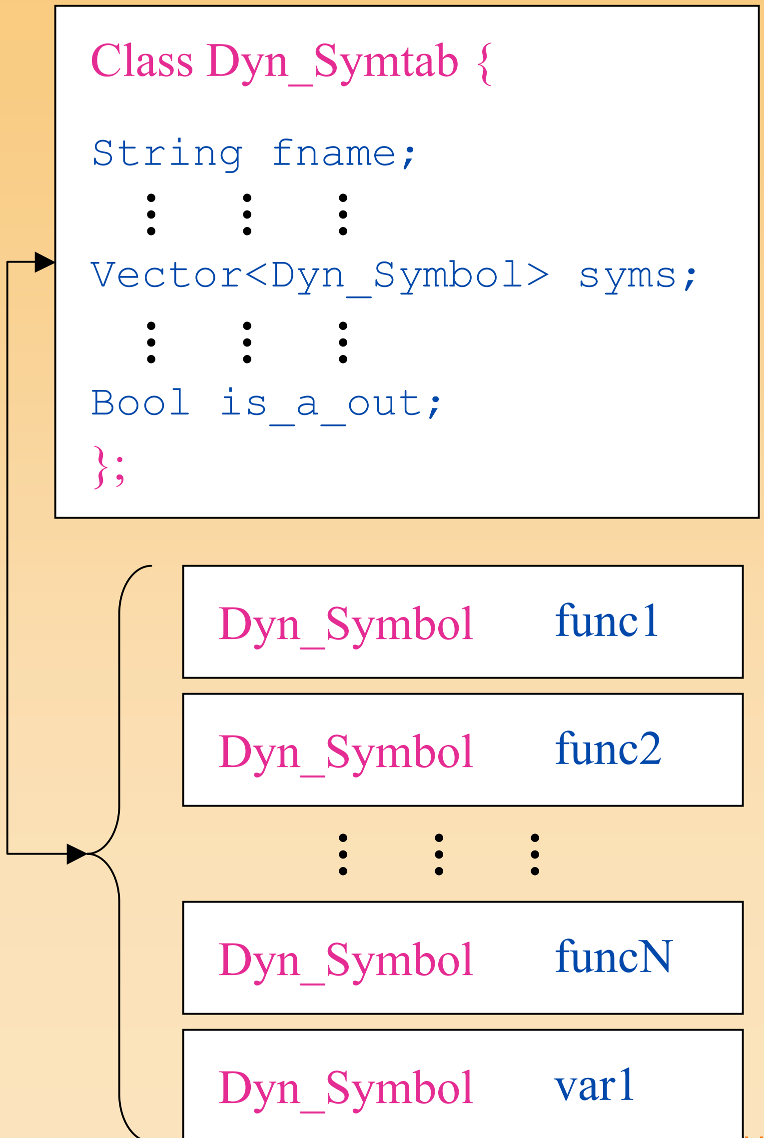


Dyn inst

# … and deserialization

- Desired behavior: A simple interface
  - deserialize(file, HierarchyRootNode,Translator)
- Requires either:
    - Alternative constructor hierarchy
      - Not consistent with extensibility requirement (need one ctor per I/O format)
    - Default constructor with subsequent setting of values
      - Functions that translate from serial stream to in-memory object
  - Child objects can be rebuilt hierarchically, but not all data structures will be saved
    - Hashes, indexing systems, etc.
    - These must be rebuilt as part of deserialization

# Simple Example Using SymtabAPI



# Simple Example Using SymtabAPI

```
Class Dyn_Symtab {
String fname;
  ⋮   ⋮   ⋮
Vector<Dyn_Symbol> syms;
  ⋮   ⋮   ⋮
Bool is_a_out;
};
```

| | |
|---|---|
| Dyn_Symbol | func1 |
| Dyn_Symbol | func2 |
| ⋮ | ⋮ |
| Dyn_Symbol | funcN |
| Dyn_Symbol | var1 |

Translator *toXML*

- open (f.xml)
- Start_symtab(f)

f.xml

<Dyn_Symtab>

Serialize( symtab, toXML, f.xml )

- Open File
- Write XML preamble

# Simple Example Using SymtabAPI

```
Class Dyn_Symtab {
String fname;
  :   :   :
Vector<Dyn_Symbol> syms;
  :   :   :
Bool is_a_out;
};
```

| | |
|---|---|
| Dyn_Symbol | func1 |
| Dyn_Symbol | func2 |
| : | : |
| Dyn_Symbol | funcN |
| Dyn_Symbol | var1 |

Translator *toXML*

- open (f.xml)
- Start_symtab(f)
- Out_val(fname)
- Out_val(is_a_out)

f.xml

<Dyn_Symtab>
<name> nm </name>
<isAOut> y </isAOut>

Serialize( symtab, toXML, f.xml )

- Write-out object fields (scalar)
- Translator can output all relevant types

# Simple Example Using SymtabAPI

```
Class Dyn_Symtab {

String fname;
  :   :   :
Vector<Dyn_Symbol> syms;
  :   :   :
Bool is_a_out;
};
```

| | |
|---|---|
| Dyn_Symbol | func1 |
| Dyn_Symbol | func2 |
| : | : |
| Dyn_Symbol | funcN |
| Dyn_Symbol | var1 |

**Translator *toXML***

- open (f.xml)
- Start_symtab(f)
- Out_val(fname)
- Out_val(is_a_out)
- Out_vector(syms)
  - Foreach (syms)
  - out_val(sym)

**f.xml**

```
<Dyn_Symtab>
  <name> nm </name>
  <isAOut> y </isAOut>
  <Dyn_SymbolList>
    <nsyms> N+1 </nsyms>
    <Dyn_Symbol>
      <name> f1 </name>
    </Dyn_Symbol>
      :   :   :
    <Dyn_Symbol>
      <name> v1 </name>
    </Dyn_Symbol>
  </Dyn_SymbolList>
```

Serialize( symtab, toXML, f.xml )

- Write-out object fields (vector)
- Helper functions take care of container classes

# Simple Example Using SymtabAPI

Class Dyn_Symtab {

```
String fname;
  :   :   :
Vector<Dyn_Symbol> syms;
  :   :   :
Bool is_a_out;
};
```

| | |
|---|---|
| Dyn_Symbol | func1 |
| Dyn_Symbol | func2 |
| : | : |
| Dyn_Symbol | funcN |
| Dyn_Symbol | var1 |

Translator *toXML*

- open (f.xml)
- Start_symtab(f)
- Out_val(fname)
- Out_val(is_a_out)
- Out_vector(syms)
  - Foreach (syms)
  - ------out_val(sym)
- End_symtab(f)
- Close(f)

f.xml

```
<Dyn_Symtab>
  <name> nm </name>
  <isAOut> y </isAOut>
  <Dyn_SymbolList>
    <nsyms> N+1 </nsyms>
    <Dyn_Symbol>
      <name> f1 </name>
    </Dyn_Symbol>
      :   :   :
    <Dyn_Symbol>
      <name> v1 </name>
    </Dyn_Symbol>
  </Dyn_SymbolList>
</Dyn_Symtab>
```

Serialize( symtab, toXML, f.xml )

- Finish up, close file



Dyn inst

# Simple Example With Binary Output

**Translator *toXML***

- open (f.xml)
- Start_symtab(f)
- Out_val(fname)
- Out_val(is_a_out)
- Out_vector(syms)
  - Foreach (syms)
  
  ------out_val(sym)
- End_symtab(f)
- Close(f)

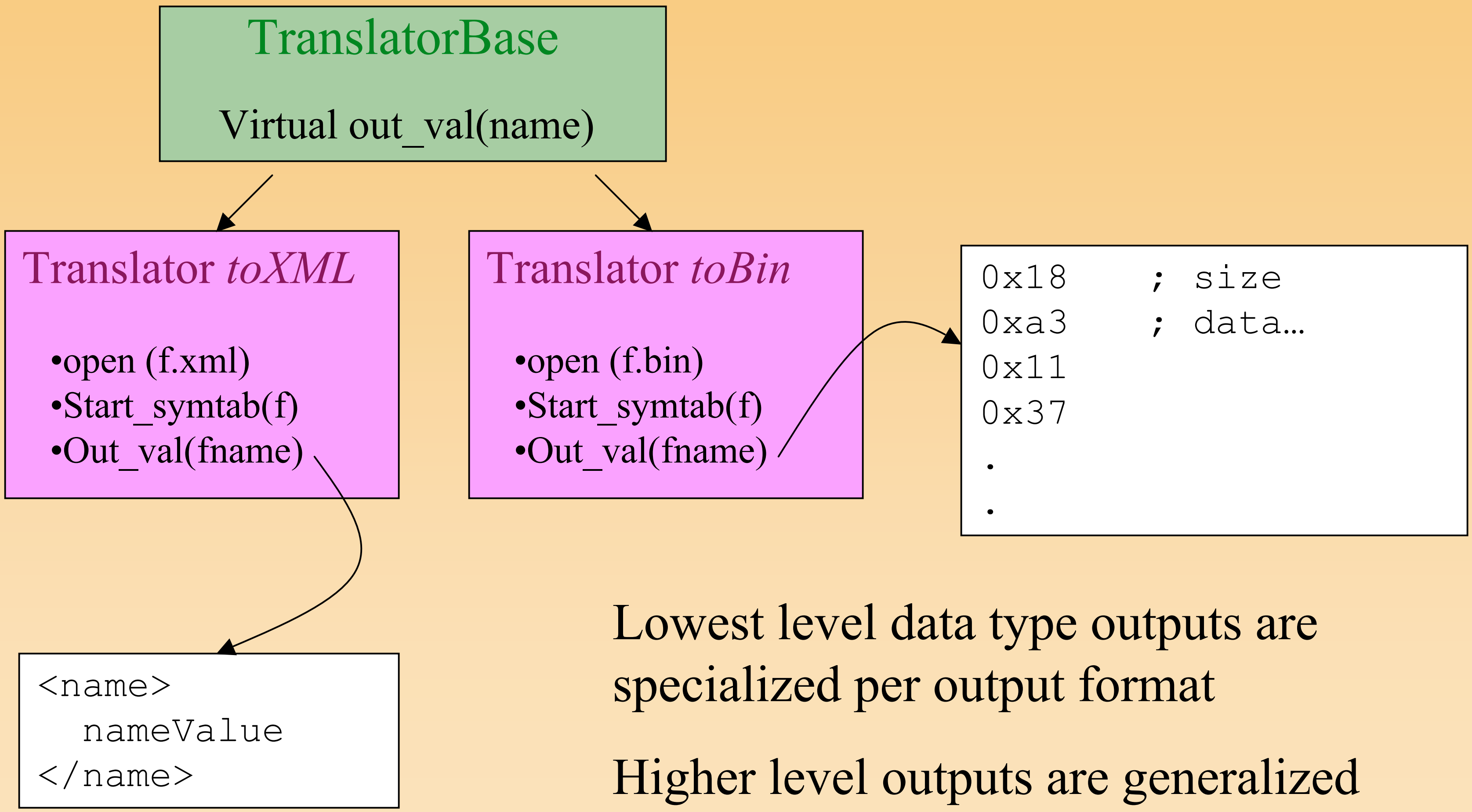
**Translator *toBin***

- open (f.xml)
- Start_symtab(f)
- Out_val(fname)
- Out_val(is_a_out)
- Out_vector(syms)
  - Foreach (syms)
  
  ------out_val(sym)
- End_symtab(f)
- Close(f)

Translator sequence is *identical*

(at the highest structural level)

# Simple Example With Binary Output




Lowest level data type outputs are specialized per output format

Higher level outputs are generalized by default, specialized as needed

# Recap

- Paradyn/Dyninst finally disentangled
  - After many years and many incremental efforts
    - (not just mine)
- Upcoming serialization / deserialization features will:
  - Improve tool performance, esp. for
    - Large binaries
    - Repeated expensive analyses
  - Allow for easier interoperability with other tools via an XML interface
    - XML spec will likely resemble the internal Dyninst class structure
    - Please contact us if you have any specific instances of interoperability we should take into account

# Questions?