UNIVERSITY OF WISCONSIN MADISON

Instrumentation Technology Update

Matthew Cheyney, Chris Serra
Brian J. N. Wylie
`wylie@cs.wisc.edu`

Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685
USA

Outline

- Current instrumentation limitations
- New technologies:
  - Multiple (local) instrumentation heap segments
  - Function relocation & expansion
  - Instrumentation of functions currently on stack
  - Resolution of statically-undetermined function calls
  - 64-bit address/instruction awareness
- Current status

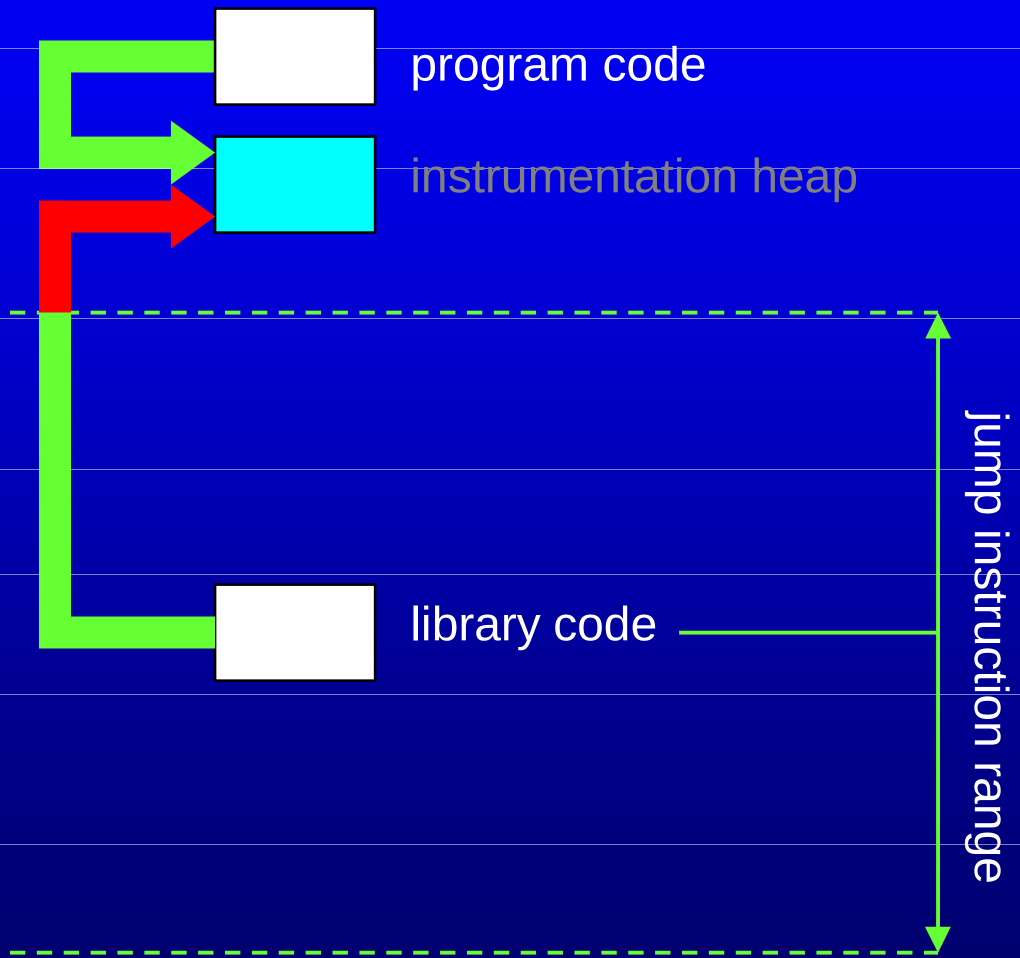
Current instrumentation limitations I

- Address spaces are too vast for 1-inst jumps
  - fast/compact jumps have insufficient reach
  - multiple instruction jump sequences required
- Some available instrumentation techniques are costly/inefficient (i.e., highly intrusive)
  - use of traps (extremely inefficient on WindowsNT)
- Some functions can't be safely instrumented in-situ (and therefore "uninstrumentable")
  - too small, too tight (highly optimized)

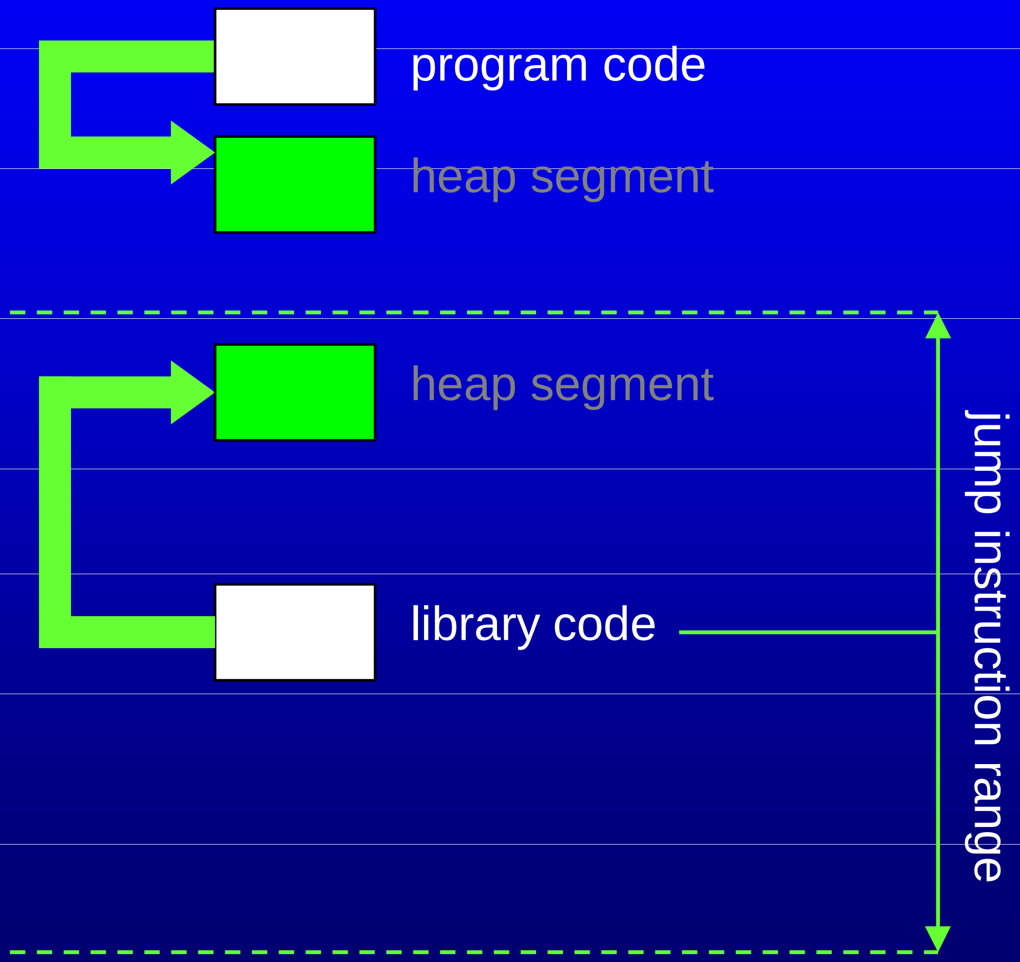
Inferior heap alternatives

- Static inferior heap [old scheme]
  - single inferior heap segment
  - statically allocated
    - implemented as large array in DynInst runtime library
- Dynamic inferior heap [new scheme]
  - multiple inferior heap segments
  - dynamically allocated in application's space
    - allocated to be near instrumentation points of interest
    - bring base-trampolines closer to instrumented code

Simple inferior heap example



Multiple inferior heap example



Dynamic inferior heap requirements

- discovery of process' address space mappings
  - **`ioctl(PIOCMAP)`**, i.e. **/proc**
- allocation of specific regions of virtual memory
  - **`mmap(MAP_FIXED)`**
- may alternatively use **`malloc()`** to allocate space within the application heap

- However, this still may not be enough
  - multiple instruction jump sequences/footprints may still be required!

Function relocation & expansion

- Copy of original function relocated to heap, selectively de-optimized, and rewritten with extra space provided for instrumentation
  - tease apart optimized call-returns (“tail-calls”) and overlapping instrumentation point footprints to allow each to be individually instrumented
  - provide extra space for footprints which overrun the end of the function or basic block
- Original function rewritten to branch to new

Reasons for relocation/expansion

1. Instrumentation footprints would overlap
2. Instrumentation footprint internally contains a branch target (i.e., crosses a basic block boundary)
3. Instrumentation footprint would extend past the end of function

- Previously, these would all have resulted in functions considered “uninstrumentable”

Relocation/expansion example

```
0x01: inst1
0x02: call A
0x03: inst3
0x04: ?br +4
0x05: call B
0x06: inst6
0x07: ret
0x08: inst8
0x09: ?br +3
0x0A: call C
0x0B: ret
0x0C: inst12
0x0D: inst13
0x0E: call D
0x0F: inst15
0x10: ret
```

Original function

Type1 +2

Footprint overlap/conflict analysis

⃠ Type2 +1

Type1 +1

⃠ Type2 +1

Type3 +1

```
0x101: inst1
0x102: nop
0x103: nop
0x104: call A
0x105: inst3
0x106: ?br +5
0x107: call B
0x108: inst6
0x109: ret
0x10A: nop
0x10B: inst8
0x10C: ?br +5
0x10D: call C
0x10E: nop
0x10F: ret
0x110: nop
0x111: inst12
0x112: inst13
0x113: call D
0x114: inst15
0x115: ret
0x116: nop
```

Relocated expanded function

Relocation/expansion process

- During object parsing, functions marked as "instrumentable-with-relocation/expansion"
  - necessary rewriting/expansion actions noted
- Relocation/expansion of function only performed when instrumentation requested
  - allows efficient use of inferior heap space
  - allows instrumentation optimization for function

Relocation/expansion benefits

- New function can be (safely) instrumented more thoroughly
  - more points (and entire functions!) become instrumentable, potentially even every instruction
- New function can be (safely) instrumented more efficiently
  - larger instrumentation footprints avoid the need to use costly traps
  - instrumentation can be “optimized” with function

Rewriting requirements

- Function expansion/rewriting must preserve execution semantics
  - retain expected order of execution
  - set context for de-optimized sequences
  - adjust branches/jumps affected by expansion and relocation of targets
- Allocate sufficient heap space for expanded function (near function or instrumentation)

Complementary solutions

- Mapping of local instrumentation heaps brings them within desired range
- Rewriting select functions with expansion provided for desired instrumentation

- More points & functions become instru'ble!
- More efficient instrumentation can be used!
  - Instrumentation optimizations become possible

Current instrumentation limitations II

- Instrumentation of functions on the stack is deferred until they return to their caller
  - ensures integrity of function instrumentation
  - often inconvenient for exclusive metrics
  - always problematic for inclusive metrics
- Some function calls cannot be determined from static analysis

Instrumentation assumptions

- Instrumentation relations:
  - entry(A) < pre-call(B) < post-call(B) < return(A)
  - pre-call(A) < entry(A) < return(A) < post-call(A)
  - no other relations supported (though definable)
- Instrumentation scenarios:
  - function is within body of stack
  - function is currently top of stack (contains %pc)
  - may have multiple instrumentation requests, each of which are processed in turn

Stack function instrumentation

- Functions currently on the stack need very careful instrumentation
  - function entry and active callee pre-call instrumentation should be executed immediately
    - use one-time-code
    - set flags, start timers, etc. (instrumentation context)
  - function return addresses on stack should be updated to return via base trampolines which contain post-call instrumentation
  - other instrumentation can be freely inserted

Body-of-stack function instrumentation

- Update context as if already instrumented
  - instrument function entry, returns and call-sites
  - immediately execute function entry-point and active call-site pre-call instrumentation
  - revise stack frame with address of active call-site location in base trampoline, so that return of callee will continue execution with post-call instrumentation

Top-of-stack function instrumentation

- Instrumentation of the function at the top of the stack (i.e., where the %pc is currently) requires additional care
  - instrument function entry, returns and call-sites
  - execute entry-point instrumentation
  - overwriting the %pc location (or relocation of the entire function) should also update the %pc

Call-stack instrumentation example

```
main()
   subA()
   subB() if (…)
   subC()
      loop
         subD1() if (…)
         subD2() if (…)
         subD3()
      until (…)
   subB()
```

Code structure

Interrupt during subD2 to instrument subC

Call stack

```
Fr. currentAddr

 0. subD2+32
 1. subC.subD2*
 2. main.subC
```

Virtual instrumentation execution record

```
main.entry
main.pre-call(subA)
subA.entry
subA.return
main.post-call(subA)
main.pre-call(subB)
subB.entry
subB.return
main.post-call(subB)
main.pre-call(subC)
subC.entry
subC.pre-call(subD1)
subD1.entry
subD1.return
subC.post-call(subD1)
subC.pre-call(subD2)
subD2.entry
…_
```

Dynamic function call resolution

- Some function calls (e.g., call-thru-register) can't be statically determined
  - call destination only determined at run-time!
  - call destination may be input-data dependent!
- Resolution requires run-time instrumentation
  - pre-instrument call-site to report the destination address found in the argument register
  - only new call destinations need to be reported

Dynamic function call resolution

Object code

```
0x28: ...
0x29: %reg=...
0x2A: call %reg
0x2B: ...
```

Trampoline pseudocode

```
destAddr=%reg;
callAddr=%PC; // 0x2A

if (destAddr ∉ visitedDests{callAddr})
    add destAddr to visitedDests{callAddr};
    report new destAddr;
fi

execute pre-call instrumentation;

call destAddr;

execute post-call instrumentation;

branch back to original code; // 0x2B
```

Run-time instrumentation benefits

- Performance Consultant bottleneck analysis (and other run-time analyses) can benefit from improved support for instrumentation
  - of functions currently on the stack (which are therefore more likely to be of interest)
  - which resolves statically-undetermined call destinations to support construction of dynamic call-graph (and graph-directed analysis)

64-bit readiness

- Address and RegValue types now used internally throughout DynInst & Paradynd
  - configurable 32- or 64-bit size
  - needs exercising on true 64-bit applications
  - need to examine mixed 32/64-bit scenarios
- 64-bit instructions and instruction “bundles” need further consideration

Current status

- Address type now used for all platforms
- Multiple inferior heap segment management implemented for MIPS-IRIX
  - further implementations just starting
- Function rewriting infrastructure implemented for SPARC-Solaris
  - thorough testing in progress
- Stack function instrumentation and dynamic function call resolution started for SPARC-Solaris

Conclusions

- App. developers are getting what they want
  - vast address spaces & more optimal (denser) code
- Tool developers aren’t getting what they need
  - improved debugging/tuning support
  - fast & compact long-range jump instructions
- Therefore
  - less code is instru’ble with existing techniques
  - more advanced instrumentation, rewriting and management techniques are increasingly required!