Memory Access Instrumentation with Dyninst

Vasile Gaburici
Motivation

- **Dynamic memory access instrumentation**
  - collect low level memory accesses
  - with the flexibility of dynamic instrumentation

- **Possible applications**
  - offline performance analysis (Sigma etc.)
  - online optimization
  - tools to catch memory errors
Features

- Finding memory access instructions
  - loads, stores, prefetches

- Decoded instruction information
  - type of instruction
  - constants and registers involved in computing
    - the effective address
    - the number of bytes moved
  - available in the mutator before execution

- Memory access snippets
  - effective address in process space
  - byte count
  - available in mutatee at execution time
Searching for memory access instructions

• **A two step process**
  - build a set of instruction opcodes to search for
  - perform the search

• **Generic opcode set specification**
  - encapsulated in `BPatch_Set<BPatch_opCode>`
  - `BPatch_opCode` (an enumeration) supports:
    - `BPatch_opLoad`
    - `BPatch_opStore`
    - `BPatch_opPrefetch`
Searching for memory access instructions [cont’d]

- **Performing the search**
  - input is an instruction opcode set
  - output is a vector of memory instrumentation points
  - search is performed at function level:
    ```cpp
    BPatch_Vector<BPatch_point*>* Bpatch_function::findPoint(
      const BPatch_Set<BPatch_opCode>& ops)
    ```
  - there is a wrapper for this at image level:
    ```cpp
    Bpatch_Image::findProcedurePoint(…)
    ```
Gathering data about accesses

- **class BPatch_effectiveAddressExpr**
  - defines an expression that returns the base address of the memory access

- **class BPatch_bytesAccessedExpr**
  - defines an expression that returns the base address of the memory access

- **Both classes have null constructors**
  - these new snippets are “point aware”
  - memory access information is extracted from the underlying memory instrumentation point
  - this simplifies usage, no need for the user to keep track and pass the correct information when the snippets are built
Example

- Instrument all loads and stores in function foo with a snippet that prints out the effective address.

```cpp
BPatch_Set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);
BPatch_Vector<BPatch_point*>* r =
    img->findProcedurePoint("foo", axs);
BPatch_Vector<BPatch_snippet*> printfArgs;
BPatch_constExpr fmt("Access at: %d.\n");
printfArgs.push_back(&fmt);
BPatch_effectiveAddressExpr eae;
printfArgs.push_back(&eae);
BPatch_function *printfFunc =
    img->findFunction("printf");
BPatch_funcCallExpr printfCall(
    *printfFunc, printfArgs);
thr->insertSnippet(printfCall, *r);```

Memory instrumentation points

- are created by memory access search functions.
- are arbitrary instrumentation points.
- provide additional memory access information
  - Bpatch_point::getMemoryAccess().
- are just like other arbitrary instrumentation points.
- are the only instrumentation points that may be used by the trace generation snippets.
Memory access information classes

• **Class BPatch\_memoryAccess contains**
  - instruction type: load, store, prefetch
    - combinations are possible (e.g. swap)
  - prefetch additional information
  - descriptor for effective address
  - descriptor for number of bytes moved

• **Effective address descriptor**
  - class BPatch\_addrSpec\_NP
  - [note: non-portable and subject to change]
  - a sum of registers plus a constant
Effective address calculation

- The memory access information is used to generate the code that computes the effective address

Address
Descriptor

R2 + R1 + C

Instrumentation

- load S, C
- _restore T,R1
- add S,S,T
- _restore T,R2
- add S,S,T

Original value obtained from:
- live register
- stack frame
- register wheel
Memory access
information classes [cont’d]

● **Byte count descriptor**
  - class BPatch_countSpec_NP
  - an alias for address descriptor right now
  - this may not hold in the future; usually the byte count has a simpler form so we may implement it differently for optimization purposes

● **Other issues**
  - register numbers are “obvious” for RISC CPUs
  - pseudo registers are used to avoid complicating the formula with bit masks used by some CPUs
    • the only one now is for PowerPC - XER 25:31
Current Status (Dyninst 3.0)

- **AIX/Power3 implementation**
  - no known limitations/problems

- **Solaris/SPARCv9 implementation**
  - no VIS, VIS2 yet [SPARCv9a,b]
  - limited support for alternate address spaces
  - not all memory inst. points are instrumentable...

- **New test program: test6**

- **Both implementations are 64-bit ready**
  - Dyninst 3.0 cannot load 64-bit images on Solaris or AIX
  - test suite uses assembler written 32-bit process that does test 64-bit instructions
Current/Future Work

• Intel IA-32 implementation is underway
  - will support opcodes up to Pentium 4
    (current Dyninst decoder only knows Pentium)
  - will require some API additions/changes:
    • memory to memory operations produce multiple targets per instruction
    • conditional moves will introduce conditional (predicated) snippets
    • some abstraction for prefetches, currently to SPARC-centric

• Building higher level libraries/tools
  - Sigma project integration