HPCTookit Update 2009

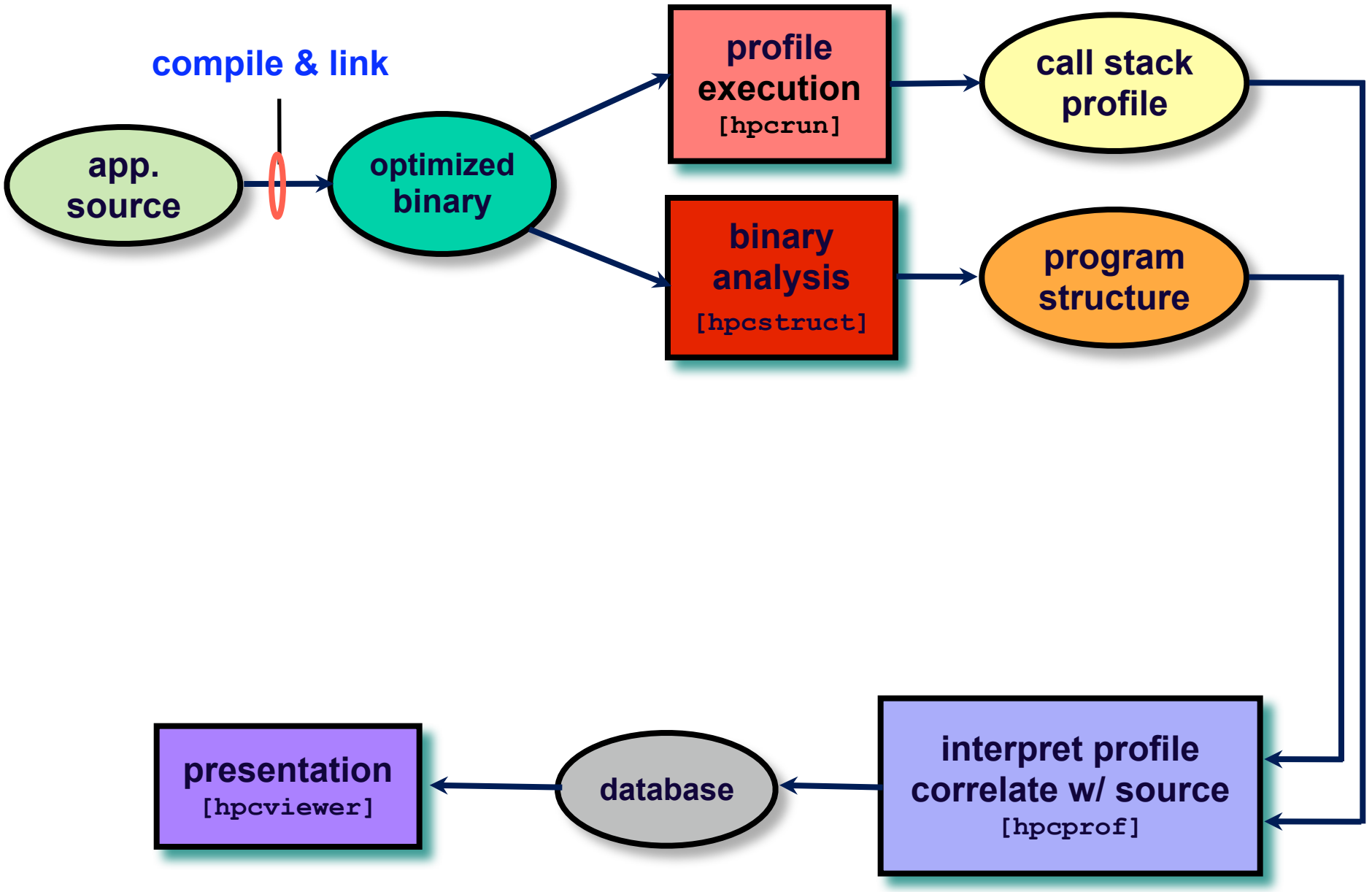
John Mellor-Crummey
Nathan Tallent
Mark Krentel
Laksono Adhianto
Mike Fagan

Rice University
CSCADS 2009

RICE

hpctoolkit.org

HPCToolkit Performance Tools



What This Talk Covers

- HPCToolkit applied to Important Applications on Leadership Class Machines
  - Includes a short demo
- HPCToolkit Stack Unwinding Technology
- Libmonitor
- Acceptance tests for sampling-based performance tools

Leadership Machines and Important Apps

- Machines
  - Jaguar
    - Cray XT4/XT5
    - National Center for Computational Science @ ORNL
  - Intrepid
    - BlueGene/P
    - Argonne National Lab
  - Both systems over 160,000 nodes
- Applications
  - MILC
    - Lattice Gauge Quantum Chomodynamics
    - Weak scaling study on both Jaguar an Intrepid
  - FLASH
    - Astrophysics thermonuclear flashes
    - Weak scaling on both Jaguar

Important Applications (cont)

- PFLOTRAN
  - Multiphase, reactive flow
  - Jaguar only
  - Strong scaling
  - Node performance via multiple metrics

Time Out ... Short Demo

- **PFLOTRAN node performance**
- **FLASH weak scaling**

hpcrun Unwind High Points

- Unwinder improved
- Unwinder has validation mode
- Implementations for:
  - —x86-64
  - —PowerPC (BG/P)
  - —MIPS (SiCortex)

Unwinding for hpcrun

- **Must work on optimized code**
  - — "frameless" procedures
  - — other non-standard prolog/epilog
- **Compute all unwind information @ runtime**
  - — Will work with dynamically loaded code
  - — No user maintenance burden
- **Fast**
  - — Lazy: compute unwind info only when actually sampled
    (cache unwind info, so computed only once)
  - — No serious control flow analysis
- **But ...**
  - — We don't have to be perfect!
  - — As long as common contexts unwind properly, dropping a rare sample is acceptable

Unwinding Methods

- **2 fundamental queries in unwinding:**
  - **Are there any more call frames? [unwind end]**
    - **hpcrun uses libmonitor for this**
  - **Given an address A, and calling context C: [unwind step]**
    - **what (A',C') pair gave rise to (A,C) ?**
      (what is the next step in the unwind ?)

- **Unwind step uses a *recipe* (= function of address&state)**

```
100: mov rax, rbx

RA = *(sp + 20)
sp = *(sp + 21)
```

```
2000: mov rax, rbx

RA = *(bp)
bp = *(bp + 1)
```

General Unwinding: Computing Recipes

- **Fundamental problem for unwind stepping is computing recipes.**
- **Key concept: use binary analysis of instructions**
- **Conceptual Algorithm**

```
Given address A
Compute RStart,REnd, the bounds of the routine containing A
// At RStart, rtn address on top of “stack”, context is known
For a in [RStart, REnd]
    analyze instruction @ a.
    compute recipe for a based on instruction semantics
      and previous recipes
      //  prev recipe: RA = *(sp)
      // 100: push rax
      //  recipe(100) ==> RA = *(sp+1)
```

Computing Recipes: hpcrun

- General unwind recipe computation:
  - —Requires A LOT of state ==> so impractical
- So, what is minimum state that will (mostly) work?
  - —Just bp (=”frame pointer”)
    - samples in prologs, epilogs FAIL
    - routines that don’t use bp FAIL (miss a frame)
    - routines that use bp as a scratch register FAIL
  - —Just sp
    - alloca or variable size local data FAIL

      ! pg implementation of alloca is a side effecting function !
- hpcrun tracks both bp & sp.
  - —each recipe tracks ra, bp, sp and which of bp or sp to use.
  - —for standard frames, we try bp first, and then sp if bp recipe fails

Computing Recipes: hpcrun (cont.)

- Routines with 1 epilog are relatively easy.
- Multiple epilogs, absent control flow analysis, require good heuristics
  - —When a ret, indirect jmp, or tail call jmp is encountered, what recipe should the following instruction use as a basis?
    - hpcrun selects one of the previously encountered recipes as a *canonical frame*
- Canonical frame heuristic
  - —If there is a previous recipe that uses bp to compute ra
  - —Find the recipe (using sp to compute ra) with the largest offset (usually means frame is completely built)
- In addition:
  - —If ret is encountered, RA recipe should be *(sp).
    If not, fixup all recipes from canonical frame choice to ret

Computing Procedure Bounds

- Computing unwind recipes requires correct function bounds
  - libraries are frequently partially stripped
    - math, communication, system
  - one bad unwind step ruins the porridge
- Our approach
  - only needs to be good enough to support unwinding
  - fast: use linear scan
- Heuristics to recover procedures in partially stripped code
  - key observation
    - some errors are tolerable
      - extend function-end to include data
    - some errors are NOT tolerable
      - clip the prolog

Computing Procedure Bounds (cont.)

- **Assumption: All procedures are contiguous**
  - **Not true: hot/cold path splitting**

    Prefer to infer 2 procedures, and make the unwind more complicated
- **Extract initial procedure information from load module (Thanks, SymtabAPI)**
  - **Global symbols are NON-removable candidates**
  - **Local symbols are still removable**
- **Linear scan through code looking for removable candidates**
  - **Address following a non-local branch (ret, uncond br)**
  - **Address after a call IFF it is a canonical function prolog**
- **Also, during the linear scan, look for instructions that cause the removal of removable candidates**
- **Remaining candidates are the function starts**

Heuristics for Removing Candidates

- If a conditional branch to t occurs @ address a:
  - The interval between a and t is a *protected* interval
    - a < t ==> [a,t') is protected
    - a > t ==> [t, a') is protected
  - All removable candidates are removed from protected interval, no removable candidates are generated in a protected interval.
- An unconditional backward branch @ addr a into a protected interval [s,e) extends interval to [s,a')
- Increment sp by L @ addr a, with corresponding decr by L at e1, en makes [a, max(e)') protected
- Interval between mov bp,sp and mov sp,bp is protected
- Interval between push bp and pop bp is protected

Unwinding Split Procedures

- **IF**
  - **Last instruction of procedure R is jmp T**
  - **Instruction just before T @ location pre(T) is jmp begin(R)**
- **THEN**
  - **Use recipe @ pre(T) as the starting point for R**
  - **Recompute all R recipes**

So, How Well Does It Work?

- **For PFLOTRAN**
  - **148 unwind failures out of 289M unwinds (8192 Processors)**
- **For our Spec benchmark test suite, compiled with Intel, PGI, and Pathscale**
  - **292 unwind failures out of 18M unwinds**

Validating Unwinds

- It is conceivable that an unwind could succeed, but not be correct.
- So, hpcrun can now (partially) validate unwind steps
  - Preliminary attempt
  - Expensive, so not for production runs.
  - Unwind steps are classified as:
    - Confirmed
    - Probable
    - Wrong

Verifying Call Stack Unwinds

- **Prove an unwind step ($f_{@callsite\text{-}x} \rightarrow g$) is possible**
  - **"Confirmed"**
    - **direct calls: $f_x \rightarrow g$**
    - **dynamically linked: $f_x \rightarrow$ [program-linkage-table] $\rightarrow g$**
    - **tail calls (1 level): $f_x \rightarrow h$ [tail call] $\mapsto g$**
  - **"Probable"**
    - **indirect calls (dynamic dispatch)**
      - **improvement: use self-modifying code to confirm at runtime**
    - **tail calls (≥ 2 levels)**
  - **"Fails"**
- **Results for SPEC / 'train' input / base + peak / Pathscale**
  - **confirmed: 7611192 indirect: 2525510 tail: 4175 wrong: 1**
  - **~59% of runs: ≥ 95% confirmed steps, 0 failures**
  - **~78% of runs: ≥ 90% confirmed steps, 0 failures**
  - **rest of the runs: 14-65% probable steps**
    - **mostly indirect calls; a few tail calls; 1 failure [?]**

What is libmonitor?

- **libmonitor is a component in the form of a library that gives access to various events of the program**
- **The API is via callbacks for the various events**
- **libmonitor gets access to the events via LD_PRELOAD**
- **hpcrun uses the monitor component extensively**

Process startup:
Monitor provides `monitor_init_process` callback

```
void *
monitor_init_process(int *,char **,void *)
{
    start_samples();
}
```

What is new in libmonitor?

- **Generic support for MPI.**
  - **This allows one monitor implementation to work with most any MPI implementation.**
  - **Downside: MPI comm size/rank is not known until the application calls MPI_Comm_rank().**
- **Overrides for the PMPI_* functions**
  - **catch MPI functions with applications that are linked with a profiling library (e.g. jumpshot)**
- **Some bug fixes**

Acceptance Tests for Sampling

- **Sampling-based tools are good stress test for system hardware/software**

- **As we deploy HPCToolkit on various leadership class machines, we are collecting a set of acceptance tests that check out systemic features that support/enable sampling-based profiling.**

Current Acceptance Tests

- sigaction returns full and correct context
- (supplied) PAPI implementation supports the sampling mode
- Sampling works with multiple threads
- Sampling is handled properly across fork/exec
- Nested signal handlers work
  - sigsegv inside a sigprof
- Signal handlers must properly restore the mask for blocked signals
- itimer with ITIMER_PROF in one-shot mode delivers the wrong signal
- Various perfctr bugs on specific Intel models
- mmap can be performed inside a signal handler