PADRONE: a Platform for Online Profiling, Analysis, and Optimization

Emmanuel Riou Erven Rohou Philippe Clauss
Nabil Hallou Alain Ketterlin

Dynamic Compilation Everywhere @ HIPEAC'14
January 21, 2014

Outline

PADRONE
- Architecture
- Usage Scenarios
- Profiling
- Analysis
- Optimization

Experiments
- Profiling Overhead
- Performance Analysis
- Function Replacement

Outline

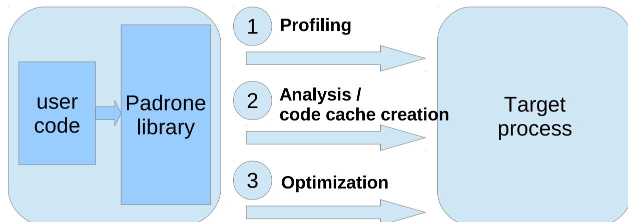
PADRONE
- Architecture
- Usage Scenarios
- Profiling
- Analysis
- Optimization

Experiments
- Profiling Overhead
- Performance Analysis
- Function Replacement

PADRONE

Architecture

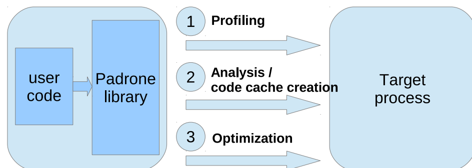
- Debugger-like interface: a *client* controls target process execution (attach, detach, manipulate)



- PADRONE: a library and API
- Leverages Linux-specific system tools (`ptrace`, `perf_event_open`, ...)
- Provides a toolbox of basic analysis/optimization operations

PADRONE

Usage Scenarios



1. Profiling only
   either global (`perf`-like), or localized
2. Temporary instrumentation/tracing/...
3. Dynamic hot-spot detection and optimization
4. Binary replacement for long-running applications/servers
   for error correction, or optimization
5. Off-line generated versions, run time selection

...

- Low-cost, efficient run-time behavior indicators
- "Performance counters"
  (avoids code instrumentation)
- Using available infrastructure (Linux)

```
int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

- Uniformly covers
  - *hardware* events: (CPU cycles, instructions, cache misses, ...)
  - *os-level* events: (context switches, CPU migrations, ...)
- Targets individual process, threads, ..., CPUs

PADRONE

Profiling

- Various modes of operation:
  - *counting* mode
    - get a file descriptor
    - read requested values *when needed*
  - *sampling* mode (frequency-based)
    - set sampling frequency (in samples per second)
    - the kernel allocates a circular buffer
    - `mmap` the file descriptor to access the buffer
    - (with variants)
- Sample structure:
  - requested counter values
  - call-chain (actually, list of return addresses)

PADRONE

Analysis

- Every profiling sample provides:
  - counter values
  - an IP, plus a call chain

  $\rightarrow$ a distribution of IPs over time (+ calling context)
- We need higher abstractions

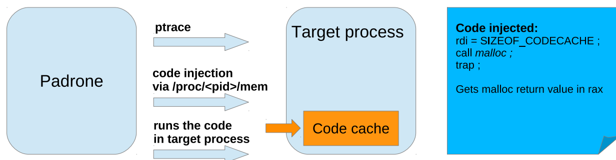
  $\rightarrow$ parsing binary code, to reconstructs regions, loops, ...
- Accessing `/proc/`*<pid>*`/{mem,maps,...}`
- Currently:
  - find enclosing function entry point
  - follow all possible control flow paths
  - find function boundaries
- Near term plans:
  - loop forest
  - symbolic access patterns
  - data & control flow analysis
  - ...

- Modifying the code of the target program...
  (by writing through `/proc/`*<pid>*`/mem`)
- Do whatever you want...

- Use `trap` (`int 3`) without moderation
  (to stop/restart the target process, leave/regain control)

PADRONE

Optimization



- Creating a code cache:
  1. stop the target process, obtain IP
  2. save code at that IP, replace with shell code
  3. restart the process, let it reach `trap`
  4. retrieve relevant registers (`malloc` return value, ...)
  5. restore original code, original IP, restart
- Writing new code: through `/proc/<pid>/mem`
- Getting this code to execute: trampolines, or patching calls

PADRONE

(could also be used for)

- Advanced (binary) code analysis
  - control-flow analysis (loops forests, ...)
  - symbolic analysis (memory access functions, loops, ...)
  - detect statically optimizable code regions
- Sophisticated program transformations
  - locality optimization
  - parallelization
- Replace loops with optimized versions

- Pradelle etal., *Polyhedral Parallelization of Binary Code*, HiPEAC & ACM TACO, 2012.

Outline

PADRONE
- Architecture
- Usage Scenarios
- Profiling
- Analysis
- Optimization

Experiments
- Profiling Overhead
- Performance Analysis
- Function Replacement

Experiments
Profiling Overhead

- Goal: measure overhead of `perf_event_open` in (*sampling* mode)
- Vary:
  - sampling frequency: 1K, 10K, and 100K per second
  - duration: 1, 10, and 30 seconds
- Results: on `444.namd`, reference time = 340

	$1K/s$	$10K/s$	$100K/s$
$1s$	343	343	341
$10s$	348	346	350
$30s$	344	356	392

(typical, with some pathological cases)

Experiments

Performance Analysis

- Goal:
  - detect the most frequently used function
  - get IPC for the next 10 calls
- Output on `470.lbm` (SPEC 2006):

```
[PADRONE] main: most frequent ip = 0x401103
[PADRONE] padrone_fun_fetch_address_from_sample:
[PADRONE]     callchain not available (no frame pointer?)
[PADRONE]     trying with libunwind
[PADRONE] padrone_fun_fetch_address:
              0x400e70 (LBM_performStreamCollide)
[PADRONE] disassembly_init: 64 bits mode
[PADRONE] ipc: 0x400e70 (LBM_performStreamCollide)
[PADRONE] ipc: ipc = 1.341582
[PADRONE] ipc: 0x400e70 (LBM_performStreamCollide)
[PADRONE] ipc: ipc = 1.345427
[PADRONE] ipc: 0x400e70 (LBM_performStreamCollide)
[PADRONE] ipc: ipc = 1.350911
...
```

Experiments

Performance Analysis

- PADRONE tool source code:

```
if (padrone_init(&padrone, pid) < 0)
        return EXIT_FAILURE;

err = padrone_profile(&padrone, duration, frequency, &prof);

mfs = padrone_profile_get_mfs(&prof);
err = padrone_fun_fetch_address(&padrone, &fun_info, mfs);

padrone_cfg_init(&cfg, &fun_info);
err = padrone_cfg_build(&cfg);

padrone_fun_ipc_measure(&fun_info, 10);

padrone_profile_destroy(&prof);
padrone_cfg_destroy(&cfg);
padrone_leave(&padrone);
```

Experiments

Function Replacement

- Goal:
  - Find most frequently used function
  - If an optimized version is available, replace original
- PADRONE tool:

```
...
uint8_t assembly_64_hw [] = ...;

padrone_cc_create(&padrone, &codecache, CC_ALLOC,
                  fun_info.ip_start,
                  sizeof(assembly_64_hw) + 10);

fun_info.cc_fun_addr
    = padrone_cc_insert(&codecache, assembly_64_hw,
                        sizeof(assembly_64_hw));

padrone_cc_fun_call_replace(&fun_info, FALSE);
...
```

Experiments

Function Replacement

- Experiment: SSE-vectorized array add... on an AVX machine
- Source code (not used):

```
...
timer_start();
for ( i=0 ; i<n ; i++ ) {
    ... vecadd(a,b); ...
}
timer_end(); printf(...);
...
```

- Experiment: 1) generate an AVX-vectorized version, and 2) replace `vecadd` during execution

```
...
redstep:  63   165.58 ms    165.58 ns per iter
redstep:  64   165.51 ms    165.51 ns per iter
redstep:  65   113.10 ms    113.10 ns per iter
redstep:  66    89.32 ms     89.32 ns per iter
redstep:  67    89.37 ms     89.37 ns per iter
...
```

Conclusion

PADRONE:

- controls a running application
- uses low-cost profiling techniques
- accesses application code for analysis
- can inject modified code

We expect to:

- provide a “complete” toolbox for common operations
- develop new program management tools
- enhance on-the-fly optimization abilities