

Automatic Low Overhead Program Instrumentation with the LOPI Framework

Simon Kågström, Håkan Grahn, Lars Lundberg
ska@bth.se

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
<http://www.bth.se/tek>

Outline

- Problem definition and background
- Instrumentation approaches
- The LOPI approach
- Optimizations performed by LOPI
- Performance evaluation
- Experiences from the implementation
- Conclusions

Problem definition

- Why instrumentation?
 - Performance measurement, security, debugging, code coverage analysis etc.
- We want low perturbation to the program behavior
 - Execution time, polluting the cache, branch prediction etc.
- With LOPI, we try to minimize the execution time
 - Common instrumentation cases have been identified
 - We have implemented optimizations for the common cases

Instrumentation approaches

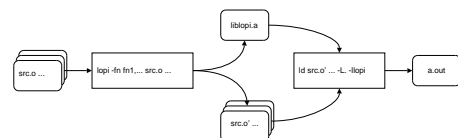
- There are many different instrumentation approaches
- Source-level instrumentation
 - Either manual or source-to-source transformations
- Binary rewriting
 - Patching the object files or the executable (LOPI)
 - Also memory image rewriting
 - Patching the program in-memory
- Other approaches
 - E.g., simulation

Instrumentation approaches, II

- Source-level instrumentation
 - Cheap (compiler-optimized), low perturbation
 - Can be tedious to perform
- Binary rewriting
 - The compiler cannot optimize
 - The source code is not needed
- Memory image rewriting
 - Similar characteristics to binary rewriting
 - Possible to insert and remove instrumentation at runtime

The LOPI approach

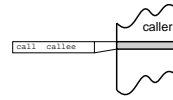
- LOPI is a package which provides a low-level interface to instrumenting applications
- The package is general and the user provides the type of instrumentation performed
- The package currently provides optimized instrumentation of function entry and exit



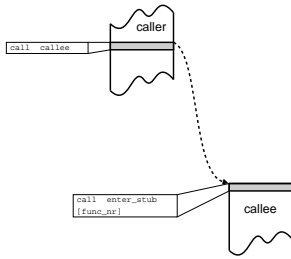
Function entry instrumentation

- Function entry is instrumented by rewriting the function prologue
- The prologue is replaced by a call to a stub which calls the instrumentation implementation and executes the overwritten instructions
- A function identifier is enclosed in the function entry
- The function entry instrumentation also handles the exit instrumentation

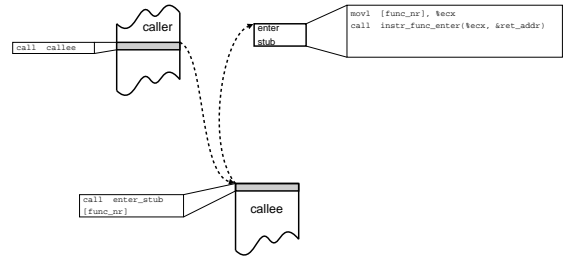
Function entry instrumentation, II



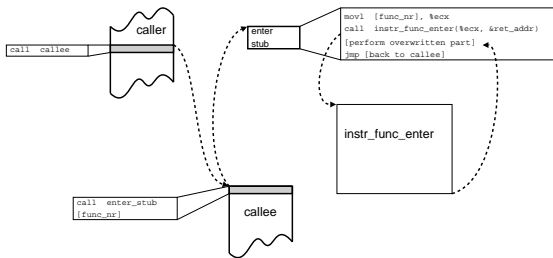
Function entry instrumentation, II



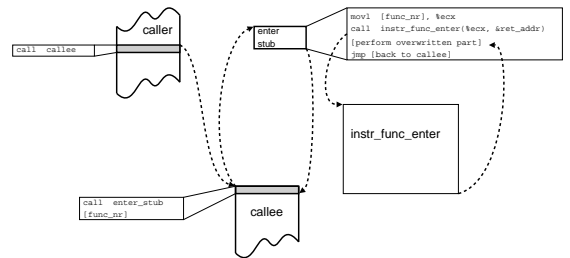
Function entry instrumentation, II



Function entry instrumentation, II



Function entry instrumentation, II



Function entry instrumentation, III

- Entering a function is instrumented as follows

```

1 typedef struct {
2     func_t *p_func;
3     long   ret_addr;
4     uint8_t program[16];
5 } ret_frame_t;
6 ret_frame_t ret_frames[];
7
8 void instr_func_enter(func_nr, ret_addr) {
9     ret_frame = pop_ret_frame(); /* Setup return frame */
10    ret_frame.func = funcs[func_nr];
11    ret_frame.ret_addr = ret_addr;
12    *ret_addr = ret_frame.program;
13
14    do_enter_func(func); /* Perform the instrumentation */
15 }

```

Knowledge Foundation <>

Automatic Low Overhead Program Instrumentation with the LOPi Framework - p. 922

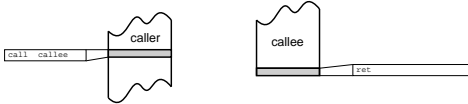
Function exit instrumentation

- Function exit instrumentation is done lazily, without rewriting the binary
- When a function is entered, the return address is overwritten with a stub-address in the return frame
- The return frame contains both code and data
- The address of the return frame stub code is used to access the data
 - Available at a fixed offset before the call

Knowledge Foundation <>

Automatic Low Overhead Program Instrumentation with the LOPi Framework - p. 1022

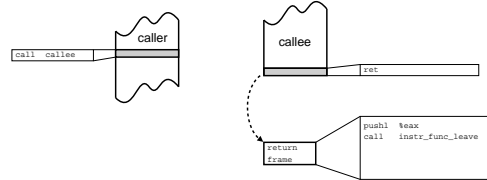
Function exit instrumentation, II



Knowledge Foundation <>

Automatic Low Overhead Program Instrumentation with the LOPi Framework - p. 1122

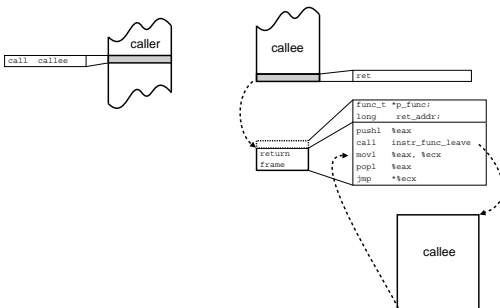
Function exit instrumentation, II



Knowledge Foundation <>

Automatic Low Overhead Program Instrumentation with the LOPi Framework - p. 1122

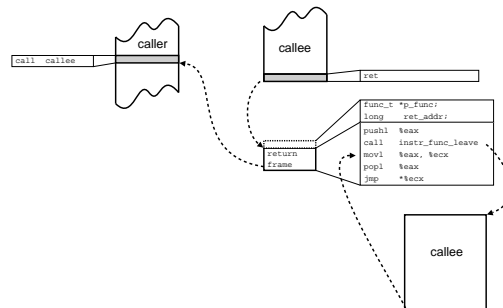
Function exit instrumentation, II



Knowledge Foundation <>

Automatic Low Overhead Program Instrumentation with the LOPi Framework - p. 1122

Function exit instrumentation, II



Knowledge Foundation <>

Automatic Low Overhead Program Instrumentation with the LOPi Framework - p. 1122

Function exit instrumentation, III

- The code for the function exit is given below

```
1 typedef struct {
2     func_t *p_func;
3     long   ret_addr;
4     uint8_t program[16];
5 } ret_frame_t;
6
7 uint32_t instr_func_leave() {
8     /* This function was called from the return stub */
9     ret_frame = [ret address]-XX;
10    /* Perform the instrumentation */
11    do_leave_func(ret_frame.func);
12
13    push_ret_frame(ret_frame);
14    return [original ret address]; /* Found in the ret_frame */
15 }
```

Example implementation

- A user implementation example is shown below
- The code accumulates hardware counters for the functions

```
1 void do_enter_func(ret_frame_t *p_f) {
2     /* Read some hardware counters into p_f->p_counters */
3     PAPI_read(event_set, p_f->p_counter);
4 }
5
6 void do_leave_func(ret_frame_t *p_f) {
7     cntr_t counter[INSO_COUNTERS];
8     int i;
9
10    PAPI_read(event_set, counter);
11    /* Accumulate the hardware counters for a function */
12    for(i=0; i<INSO_COUNTERS; i++)
13        p_f->p_func->accum[i] += counter[i] - p_f->counter[i];
14 }
```

Optimizations

- We have a special entry stub for stack frame setup

```
1 pushl %ebp # Save the old frame pointer
2 movl %esp, %ebp # Set the start of the new frame
3 subl $XX, %esp # Allocate stack space
```

- This only varies with the constant XX for all functions
- Very common (around 80% of the functions in our benchmarks)
- We store XX in the instrumentation structure for the function (func_t)
- Identical entry stubs are not duplicated

Optimizations, II

- The return frames are ordered as a stack to encourage reuse
- We apply knowledge about live registers at function entry
 - caller-saved registers can be used at will
- The function identifier is inlined in the function prologue
 - I.e., the identifier resides directly after the call-instruction
 - We can therefore do a cheap array-indexing to find the instrumentation data

Evaluation

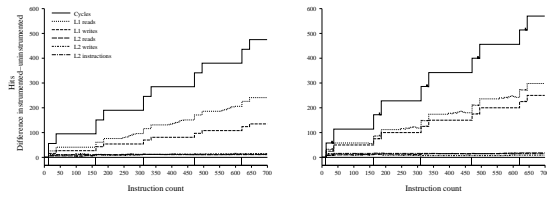
- We evaluated LOPI against source instrumentation and Dyninst
 - Real hardware: Pentium III 1GHz
 - Full-system simulator: simics
- We evaluated seven SPEC CPU2000 benchmarks: 164.gzip, 176.gcc, 181.mcf, 183.equake, 197.parser, 256.bzip2, 300.twolf
- In these, the functions responsible for 80% of the execution time was selected for instrumentation (from a gprof run)

Evaluation, II

- Source instrumentation caused least perturbation
- Compared to Dyninst, LOPI executed fewer instructions and had fewer cache references
 - But more mispredicted branches because of the overwritten return address
 - It also had slightly more cache misses than Dyninst
- On average, the LOPI execution overhead was 22% compared to 28% for Dyninst
 - In terms of cycles executed

Evaluation, III

- We also made detailed execution traces of 183.equake using Simics
- Below, we show the cache accesses made by both LOPI (left) and Dyninst (right)



An unexpected effect

- We first evaluated LOPI in a simulated environment, where it performed better than Dyninst
- To our surprise, it was worse than Dyninst on real hardware
- We first suspected branch mispredictions, but it turned out to be a lcache-Dcache conflict
 - On real hardware, instrumentation increased lcache misses a factor of 1000
- Each return frame contains both code and data
 - Return address, function info pointer
 - The return stub

An unexpected effect, II

- Easy fix: pad ret_frame to cache-align the stub code
- ```
1 typedef struct {
2 func_t *p_func;
3 long ret_addr;
4 uint8_t padding0[XX]; /* For conflict reduction */
5 uint8_t program[16];
6 uint8_t padding1[XX];
7 } ret_frame_t;
```
- Fewer misses than Dyninst again, also on real hardware

## Conclusions

- We have presented the LOPI framework for binary instrumentation
- LOPI currently instruments function entry and exit
- We have identified and provided optimizations for common cases
  - Reuse of instrumentation stubs when possible
  - Data reuse
- Our evaluation shows fewer instructions and fewer cache accesses than Dyninst, at the cost of slightly more branch prediction misses
- During the process, we learned a thing or two about the IA-32 caches

## Questions

Questions?