An Extended GPROF Profiling and Tracing Tool for Enabling Feedback-Directed Optimizations of Multithreaded Applications

S.Bartolini
Department of Information Engineering
University of Siena, Italy

C.A. Prete
Department of Information Engineering
University of Pisa, Italy

GREPS Workshop (PACT '07) – Brasov, Romania. 16/09/2007

Motivation

- Parallel architectures are enforcing the need of managing parallel software efficiently
  - Sw design, programming, compiling, **optimizing**, running
- Need to provide application-behavior feedback to compiler/linker to enable advanced optimizations
  - *e.g.*, L1 instruction cache opts through procedure remapping
    - Monothreaded application: locality features (call graph)
    - This simple information can be easily available, *e.g.* GNU GCC
    - More sophisticated information is *still not widely available*
- The behavior of a parallel (multithreaded) application is harder to profile
  - Multiple concurrent, interacting execution activities with possible dynamic activation
  - Interaction typically mediated by the Operating System
  - Complex applications → runtime profiling → overhead, probe eff.

Motivation

- Challenge: profile multithreaded (MT) applications
  - enable the extension of existing feedback-directed compiler/linker opts, and, possibly, the design of new ones
- Desiderata:
  - Easy to perform: Gprof-like user interface
  - collect simple (“Gprof”-like) stats on a per-thread basis
  - Additional intra-thread stats
    - Usage of synchronization mechanisms/API
  - Collect inter-thread stats
    - Execution ordering (cooperation)
    - Blocking relationship, blocking time (competition)
    - Idle time (competition for Hw/Os resources)
  - Low overhead
    - Enable profiles from real-execution

Outline

- GPROF intro
- Extensions for multithreading
- OS activity
- Synchro-tracing
- Results

Gprof features

- Gprof capabilities
  - *Flat-profile*: time profile of functions
    - Not very precise and not suitable for parallel applications
  - *Function call graph*
    - Locality
  - *Basic block execution count*
    - Quite expensive in terms of overhead
  - Modular approach:
    - Selectable profiling level: *overhead vs. detail*
  - Usage:
    - Compile programs (or modules) with a special GCC flag (*-pg*)
      - libraries
    - Execute the application → profiling output file
    - Gprof tool to analyze the output

Gprof example

```
index % time self children  called     name

[1] 100.0    0.00 0.05         start [1]
             0.00 0.05    1/1    main [2]
             0.00 0.00    1/2    on_exit [28]
             0.00 0.00    1/1    exit [59]
-----------------------------------------
             0.00 0.05    1/1    start [1]
[2] 100.0    0.00 0.05    1    main [2]
             0.00 0.05    1/1    report [3]
-----------------------------------------
             0.00 0.05    1/1    main [2]
[3] 100.0    0.00 0.05    1    report [3]
             0.00 0.03    8/8    timelocal [6]
             0.00 0.01    1/1    print [9]
             0.00 0.01    9/9    fgets [12]
             0.00 0.00   12/34   strncmp <cycle 1> [40]
             0.00 0.00    8/8    lookup [20]
             0.00 0.00    1/1    fopen [21]
             0.00 0.00    8/8    chewtime [24]
             0.00 0.00    8/16   skipspace [44]
```

Gprof limitations for multithreaded appl.

Call graph gathering:

- *Instrumentation and Event count*
  - Compiler adds a function call (to *mcount*()) at the very start of each function
  - *mcount* is responsible of "profiling" its execution
    - *e.g.* call-graph update: parameters(caller addr., callee addr.)
- Only the main thread is profiled
  - "main" might do only initializations
    - Possibly short and not representative behavior
- One profiling structure
  - Life-cycle (allocation, deallocation and write to file)
    - Performed at application start/end

Proposed extensions for multithreading

- One profiling structure for each thread
  - Allocation:
    - Part is pre-allocated in "chunks", on demand, in advance
      - The association to the thread is done at thread creation time (fast, flexible)
  - Flush
    - By *mcount* itself, whenever the thread-specific structure is full
      - Disable bit … per thread, to avoid *mcount* recursion
  - Deallocation
    - Upon thread termination
- Actions:
  - thread "create" and "finalize" patched similarly to "process create" and "process exit" in native Gprof
  - Altered *mcount* to support MT

Multithreaded extensions: event record

- Call graph
  - Relatively fixed size during execution
- Event history
  - Sequence of pre-defined and/or programmer specified events, in program execution order
    - *e.g.*: execution of specific instructions (like C statements)
    - *e.g.*, Track locality and/or usage of code sections
  - Event record:
    - {event_type, event_id, timestamp}
  - Each record has a timestamp
    - e.g.: TSC (Intel) or other mechanism
  - Size increases with execution time
    - Flushing needed

OS activity

- Context switch between thread-1 and thread-2 ?
  - The event log is not enough
  - *e.g.*: OS time quantum expires, blocking operation, preemption, …
- A thread cannot profile its behavior when it is not executing
  - Only indirect, sometimes misleading information
- Information from the OS needed
  - Linux Tracing Toolkit: is able to detect and log a number of events with limited overhead (less than 2.5%) and is easily customizable
    - Syscalls, interrupts, fork, scheduling, filesystem ops, virtual mem ops, network
  - We specialized LTT tracing
    - Ability to limit to the application-related events
    - Added a timestamp (same format as for intra-thread events)

Example of out LTT output

```
Event              Time          PID   Length Description
#####################################################################
Syscall exit     1042493319696474  N/A   7
Syscall entry    1042493319696492  N/A   12 SYSCALL : open; EIP : 0x0804A509
File system      1042493319696637  N/A   20 START I/O WAIT
Sched change     1042493319696696  1191  19 IN : 1191; OUT : 1215; STATE : 2
File system      1042493319696707  1191  20 SELECT : 3; TIMEOUT : 0
File system      1042493319696713  1191  20 SELECT : 4; TIMEOUT : 0
File system      1042493319696716  1191  20 SELECT : 6; TIMEOUT : 0
File system      1042493319696718  1191  20 SELECT : 7; TIMEOUT : 0
File system      1042493319696722  1191  20 SELECT : 9; TIMEOUT : 0
File system      1042493319696724  1191  20 SELECT : 11; TIMEOUT : 0
Memory           1042493319696737  1191  12 PAGE FREE ORDER : 0
Syscall exit     1042493319696745  1191  7
Syscall entry    1042493319696994  1191  12 SYSCALL : read; EIP : 0x0804D028
File system      1042493319696997  1191  20 READ : 11; COUNT : 4096
Syscall exit     1042493319697012  1191  7
Trap entry       1042493319697722  1191  13 TRAP : page fault; EIP : 0x4126C309
Trap exit        1042493319697729  1191  7
Syscall entry    1042493319698007  1191  12 SYSCALL:gettimeofday;EIP: 0x0804D028
Syscall exit     1042493319698010  1191  7
...
File system      1042493319722332  1037  20 SELECT : 17; TIMEOUT : 12000
Timer            1042493319722334  1037  17 SET TIMEOUT : 12000
Sched change     1042493319722337  1191  19 IN : 1191; OUT : 1037; STATE : 1
File system      1042493319722340  1191  20 SELECT : 3; TIMEOUT : 2
...
Syscall entry    1042493319722926  1191  12 SYSCALL : write; EIP : 0x0804D028
File system      1042493319722927  1191  20 WRITE : 3; COUNT : 8
Socket           1042493319722929  1191  16 SO SEND; TYPE : 1; SIZE : 8
Process          1042493319722935  1191  16 WAKEUP PID : 1037; STATE : 1
Syscall exit     1042493319722939  1191  7
Sched change     1042493319722942  1037  19 IN : 1037; OUT : 1191; STATE : 0
File system      1042493319722946  1037  20 SELECT : 1; TIMEOUT : 12000
```

Tracing of synchronization events

- LTT is able to show the low-level, basic synchro-ops on which the OS relies
  - Low-level mechanisms (far from programmer ones)
  - Potential variability across OS versions
- The programmer is interested in analyzing, profiling, inspecting the synchronization mechanisms he has in his scope
  - Semaphores, mutexes, condition variables, spinlocks, signals
  - …and going into the specificity of their semantics
  - Whatever their implementation inside the OS
- Aims:
  - Performance profiling (bottlenecks)
  - Behavior assessment (testing) and verification
  - Debugging
  - Execution patterns
- Action: special events recorded in the thread event log
  - Synchro API invocation trigger event log
  - Modularity and flexibility → original API wrapped inside the logging ones
- Event record:
  - *{Syncr.Type, Structure-Id (address), Op.Type, timestamp}*

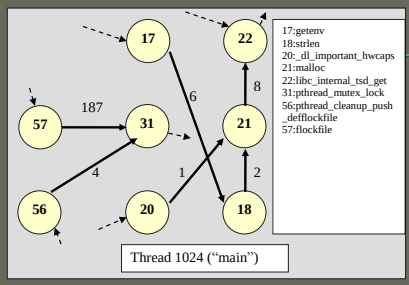
Putting it all together

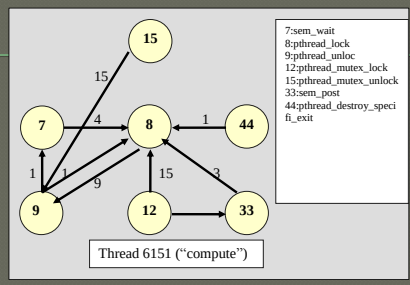
- Information gathering
  - Intra-thread: statistics and events
  - Inter-thread: thread context switches with motivation (OS), other application-related OS events (*e.g.*: scheduling, blocking, page faults)
  - Synchronization events per thread in program order
- Independent logging of different information
  - Limited probe effect on parallel behavior
  - Low overhead (time and memory)
- Post-processing of the independent stats/log files
  - Possibility to dig into the data and extract complex information
    - e.g.: semaphore usage pattern
- Low overhead is not magic :)
  - Coarse grain data collection
  - Independent profiling activities
  - Carefully designed profiling data structure

Results – test setup

- "Kernel" benchmarks
  - Dining philosophers (10 → 100 load test)
  - Simulation of vehicles in a crossroad (30 → 1000 cars)
  - Client-server application (30 → 80 concurrent clients)
  - Processor farm for matrix multiplication (6x6-6 threads → 20x20-25 threads)
- Real multithreaded application MySQL
  - Test procedures coming along the installation bundle: *test-check, test-lock, test-myisam*
- A couple of profiling structure size
  - *profile-A* = 1 M-entry (event record) per thread, 500 pre-allocated structs
  - *profile-B* = 100 entry per thread, 10 pre-allocated structs
- Various levels of profiling
  - *applic-only*: only thread stats/events are collected
  - *full-sched*: MT application + OS context-switch events
  - *full*: MT application + full OS events

Intra-thread



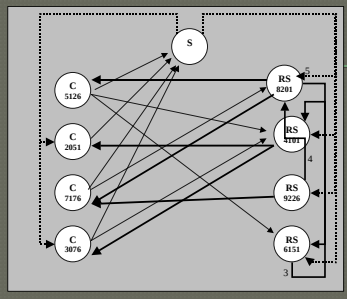


Function name	Execution count
***Thread_id* 4101**	
_IO_new_file_xsputn	**13**
__errno_location	**12**
__pthread_lock __pthread_unlock	**11**
***Thread_id* 1024**	
Sem_getvalue	**151725**
_IO_new_file_xsputs	**640**
_pthread_cleanup_push_defflockfile	**189**

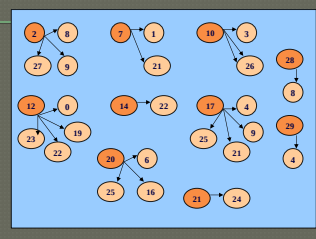
Intra-thread stats:

- *(top)* Call graphs: interaction between functions
  - *Processor farm*: small execution
- (*left*) function execution count:
  - *Processor farm*: load-condition
- Pettis&Hansen 1990 I-cache opt relies on this
  - Already in gcc for mono-threaded appl.

Inter-thread and synchro

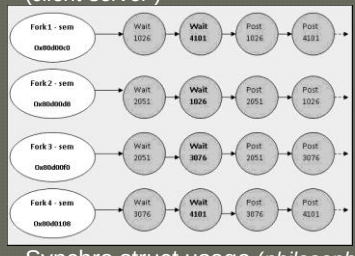


Inter-thread graph (thread-granul.) (*client-server*)



	Thread	Funzione
16	1026	pthread_start_thread
17	1026	__nanosleep
18	1026	vfprintf
19	2049	__pthread_manager
20	2049	clone
21	2049	kill
22	2049	__errno_location
23	2049	__libc_read
24	2049	poll
25	2051	pthread_start_thread
26	2051	__pthread_lock
27	2051	printf
28	2051	__errno_location
29	2051	__nanosleep

Inter-thread graph (function-granul.) (*crossroad simulator*)

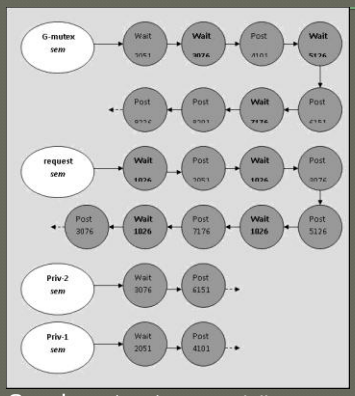


Synchro struct usage (*philosophers*)

Function/Thread	4101	1026	2051	3076
MutexLock	3/0	3/0	3/1	3/0
MutexUnLock	3/-	3/-	3/-	3/-
Wait	2/2	2/1	2/0	2/1
Post	2/-	2/-	2/-	2/-
Lock Destroy	1/-	1/-	1/-	1/-

Synchro struct usage count (*philosophers*)

Inter-thread and synchro *cont.*



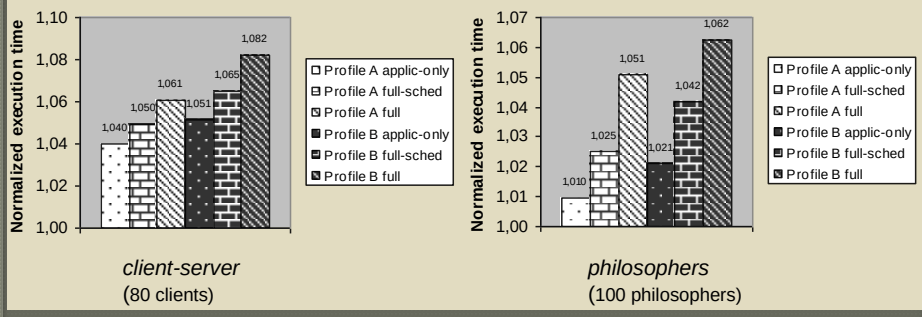
Synchro struct usage (*client-server*)

Thread_id	Not-running time (%)	Not-running time (blocked) (%)
1026	72	30
2051	63	20
3076	71	31
4101	75	37

Thread non-running time (*philosophers*)

- Ability to easily derive a variety of stats

Results: overhead

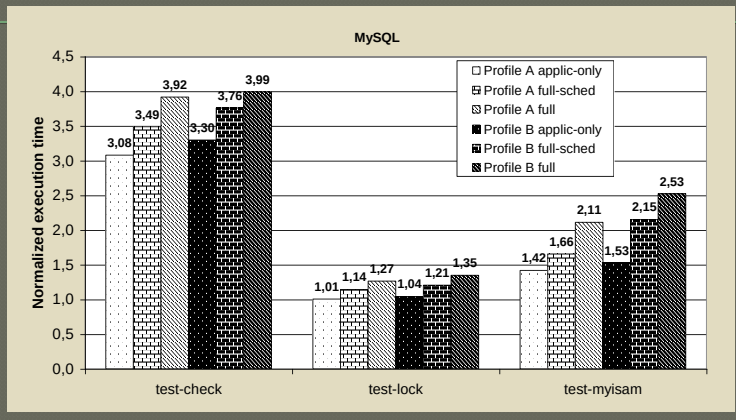


client-server (80 clients)

philosophers (100 philosophers)

- Overall overhead is reasonable for runtime profiling
- Not much degradation for *full* vs. *applic-only* profile
- Relative overhead of small profile structure (*B*) vs. large (*A*) can be high
  - But the absolute overhead value is still reasonable

Results: overhead *cont.*



- The overhead of a complex, real application can be higher
- Still reasonable for real-execution runtime profile

Conclusions

- Highlighted the limitations of Gprof in case of multithreaded applications
  - per-thread information
  - Thread relationship
  - Thread-related OS activity
  - Profile of programmer-level synchronization mechanism
- Proposal of a MT runtime profiling approach
  - Gprof inspired code instrumentation
  - Independent gathering of different information
    - Application threads, OS
    - Extension of LTT tool for Linux OS profiling
  - Post-processing of raw, low-level profile to derive high-level information
    - Intra-thread call graph, inter-thread call graph, synchro-structure usage pattern, thread idle/blocking time, …
  - Low-overhead: wide applicability
  - Suitable to enable feed-back directed compiler optimizations

Thanks for your time

Q & A ?