

University of
Waterloo



Advancing High-Performance Applications on Linux Itanium

<http://gelato.uwaterloo.ca>

SMP Concurrent Software Development

Peter Buhr (faculty)
Richard Bilson (staff)
Ashif Harji (Ph.D.)
Roy Krischer (Ph.D.)
Justyna Gidzinski (M.Math)
Ayelet Israeli (M.Math)

High-Performance Web Servers

Tim Brecht (faculty)
Mark Groves (staff)
Gary Yeung (M.Math)

Research Sponsors



Outline

1. μ C++ : concurrency in C++ (review)
2. μ Profiler : profiling μ C++ applications
 - Execution-State Transition (EST) metric
 - Call Graph (CG) metric
3. Compare μ Profiler with other concurrent profilers

μ C++ : Concurrency in C++

- Programmers want to use C++ for HPC **but C++ has no concurrency!**
- μ C++ provides high-level, integrated, object-oriented concurrency
 - coroutine, monitor, task : inheritance, overloading, templates
 - concurrent exception handling and cancellation (among threads)
 - *control* versus data concurrency (Intel TBB, OpenMP, etc.)
- Simplifies building efficient, correct concurrent-programs
- Reduces cost of development, both in time to completion and necessary intellectual skills

Can't sell parallel computers if developers can't program them.

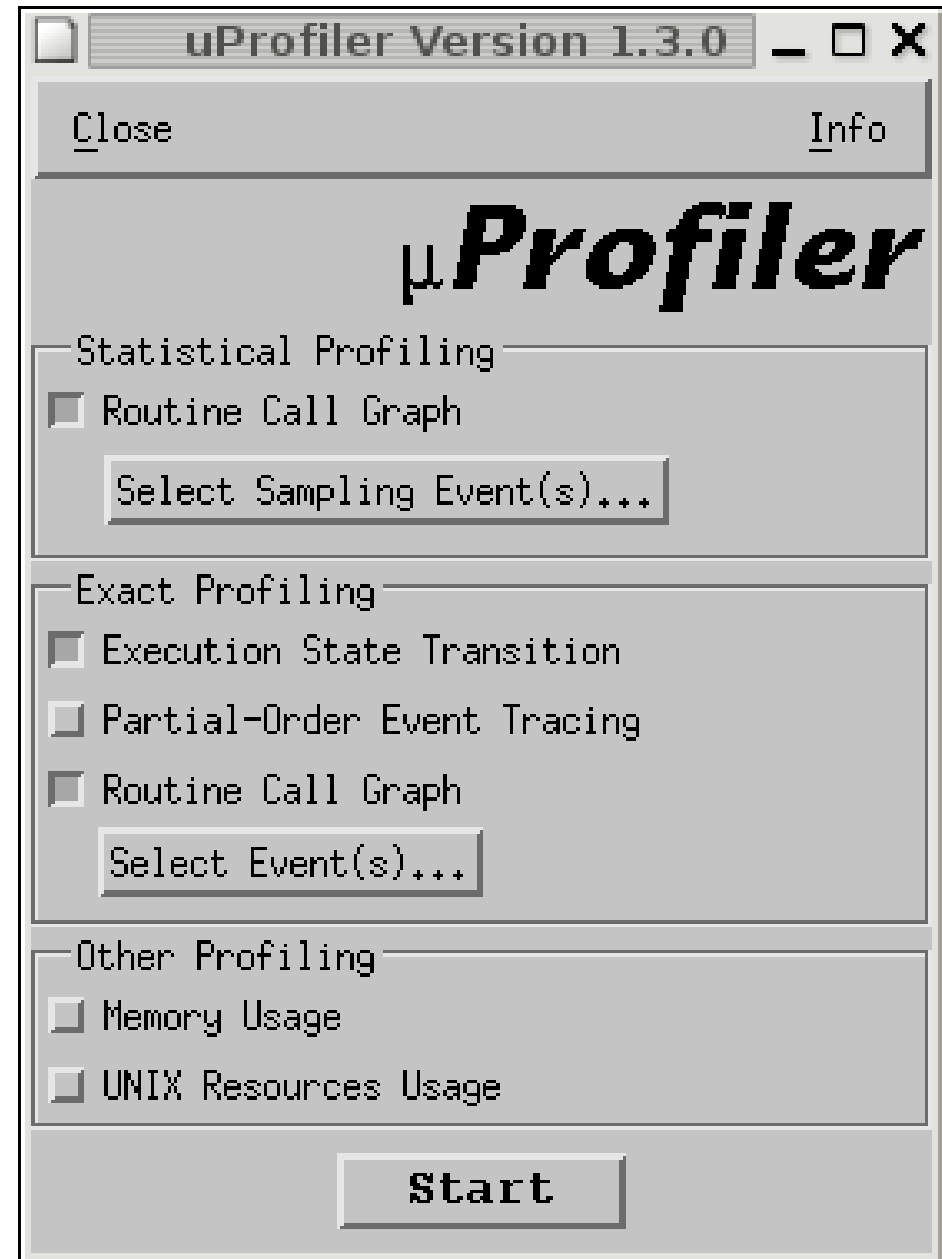
- <http://plg.uwaterloo.ca/~usystem/uC++.html>

Coroutine	Task
<pre> // 0, 1, 1, 2, 3, 5, 8, 13, 21, ... coroutine fibonacci { int fn; // <i>communication</i> void main() { // <i>separate stack</i> int fn1, fn2; fn = 0; fn1 = fn; suspend(); // <i>last resume</i> fn = 1; fn2 = fn1; fn1 = fn; suspend(); // <i>last resume</i> for (;;) { fn = fn1 + fn2; fn2 = fn1; fn1 = fn; suspend(); // <i>last resume</i> } } public: int next() { resume(); // <i>last suspend</i> return fn; } }; </pre>	<pre> task Server { condition delay; void main() { // <i>stack & thread</i> for (;;) { accept(~Server) { break; } or accept(workReq1) { // <i>process work</i> delay.signalblock(); } or accept(workReq2) { ... } } // <i>shutdown</i> } public: void workReq1(Req1 req) { ... delay.wait(); ... } void workReq2(Req2 req) { ... } ... }; </pre>

μ Profiler: Profiling in μ C++

- To further simplify developing concurrent programs, need powerful static and **dynamic** tools
- Programmer intuition about dynamic behaviour is often wrong:
⇒ need profiler to understanding concurrent execution
- Integrate profiler with programming model, for μ C++ ⇒
 - break down per-task, per-coroutine, (per-object), per-routine
- μ Profiler capabilities:
 - multiple metrics can be applied simultaneously
 - supports hardware event-counters (PMU)
 - effective, efficient, scalable and extensible
- Monitoring, analysis and visualization performed in the application
- ports: **Linux IA-64**, Linux IA-32/AMD, Solaris 8/9 SPARC
- <http://plg.uwaterloo.ca/~usystem/MVD.html>

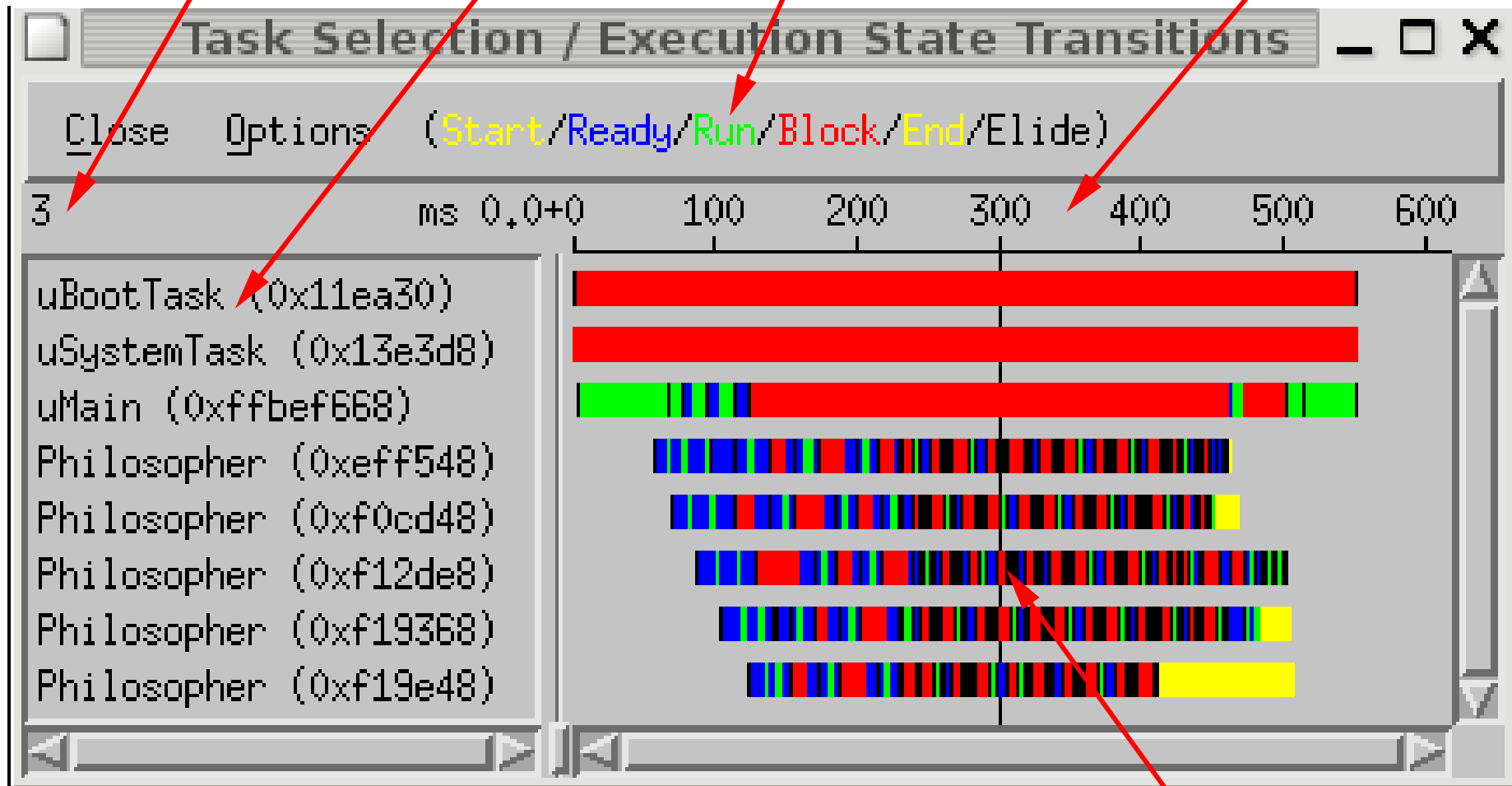
- compile with `-profile` flag
- start application
- choose metrics:
 - statistical (sampling):
 - * **call-graph**: time / hardware events
 - exact (all occurrences):
 - * **execution state transition**
 - * **call-graph**: time / hardware events
 - * partial-order tracing
 - * memory leaks
 - * kernel-threads
- press **Start**



Execution-State Transition Metric

- Per task display of all state transitions using a Gantt Chart
- Colour coded execution states: start, ready, running, blocked, end
- In general, cannot display entire chart due to duration and precision
⇒ only draw visible portion of the chart at a given magnification
 - support long program duration and high magnification (up to 0.1ns per pixel)
 - implementation avoids processing all the data during magnification and movement
- Navigation is crucial : millions/billions of state transitions occurring quickly ($< \mu\text{sec}$)

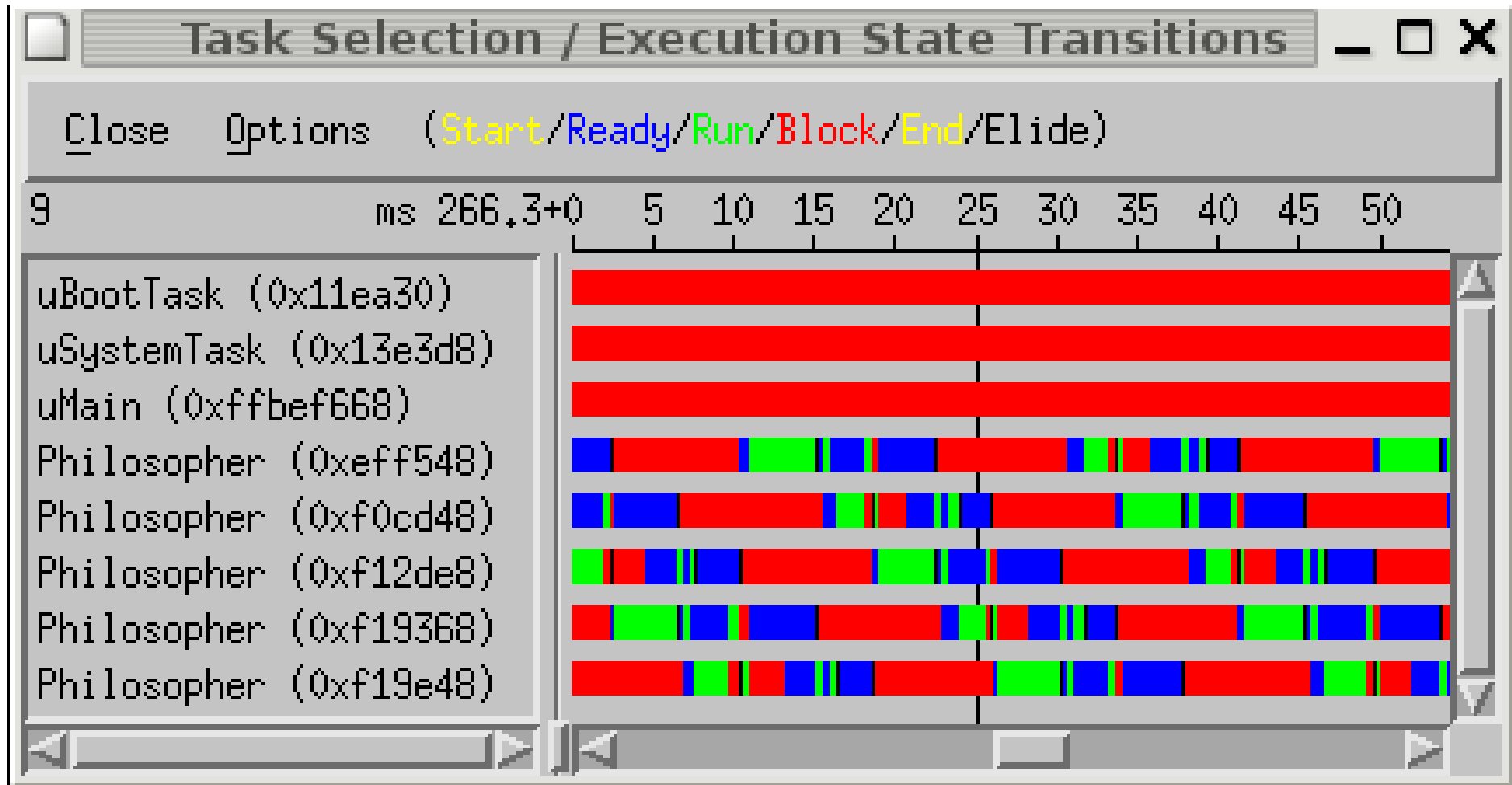
magnification task names task states dynamically scaled axis



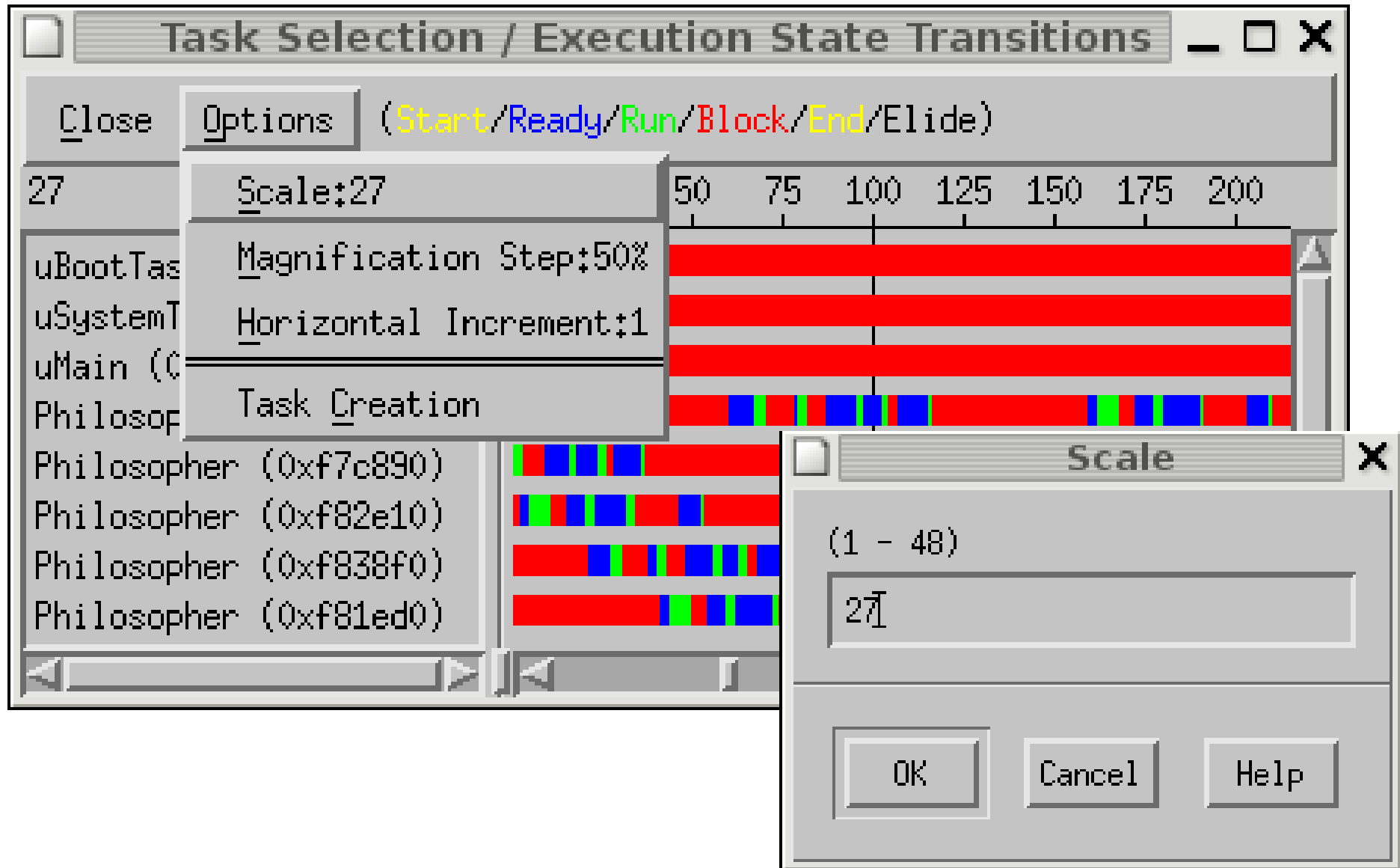
lower magnification

elided states

- elided states : compressed information
- magnification : step and scale factor
- dynamically scaled axis



higher magnification



Execution State Data : Task Philosopher (0xf7c850)

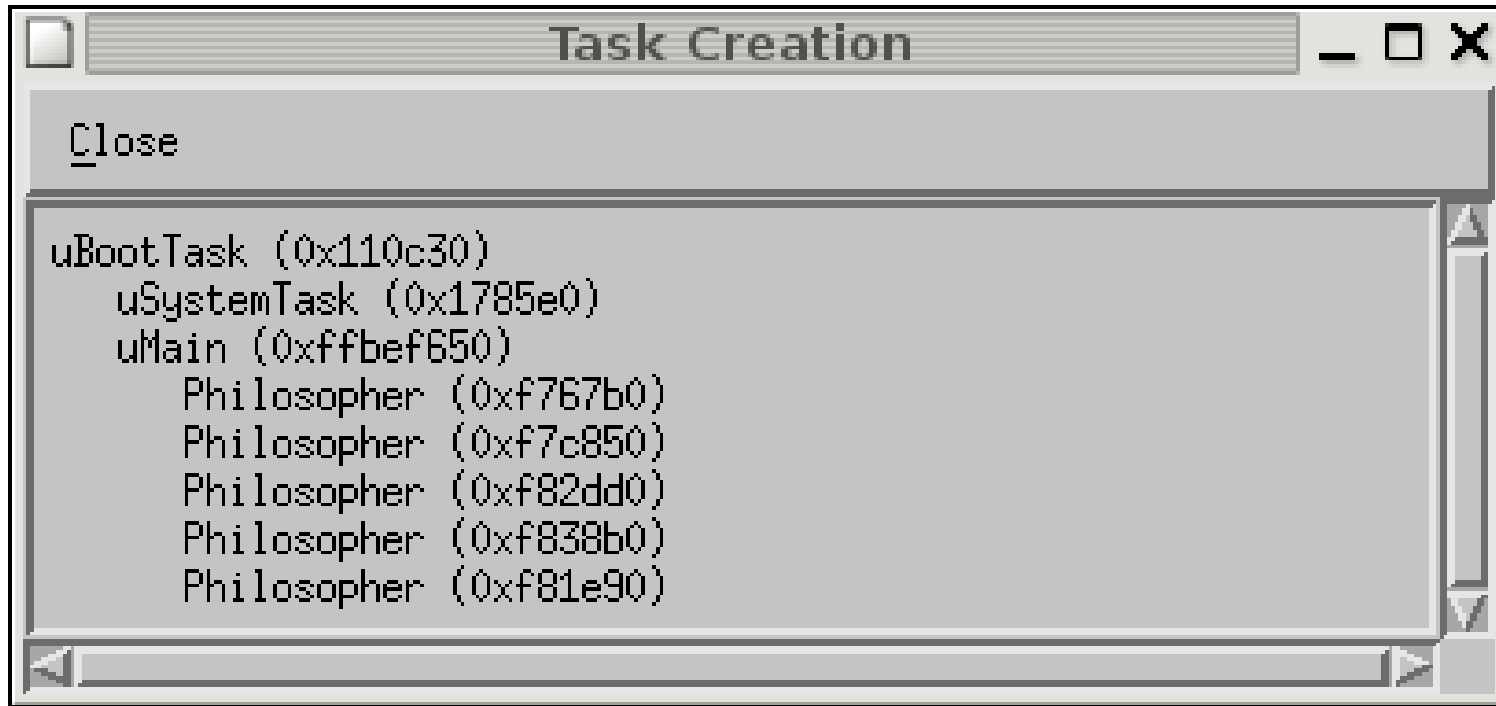
Close Options

Execution Summary

Life Time:	(msec)	(%)	Clock Time:	(H:M:S.MS.US)
ready	3242.755	100.00	Creation	08:56:14.431.781
running	877.418	27.06	Deletion	08:56:17.674.536
blocked	403.202	12.43		
State Duration:	1961.546	60.49		
Minimum	0.001			
Maximum	1.252			

State Transitions

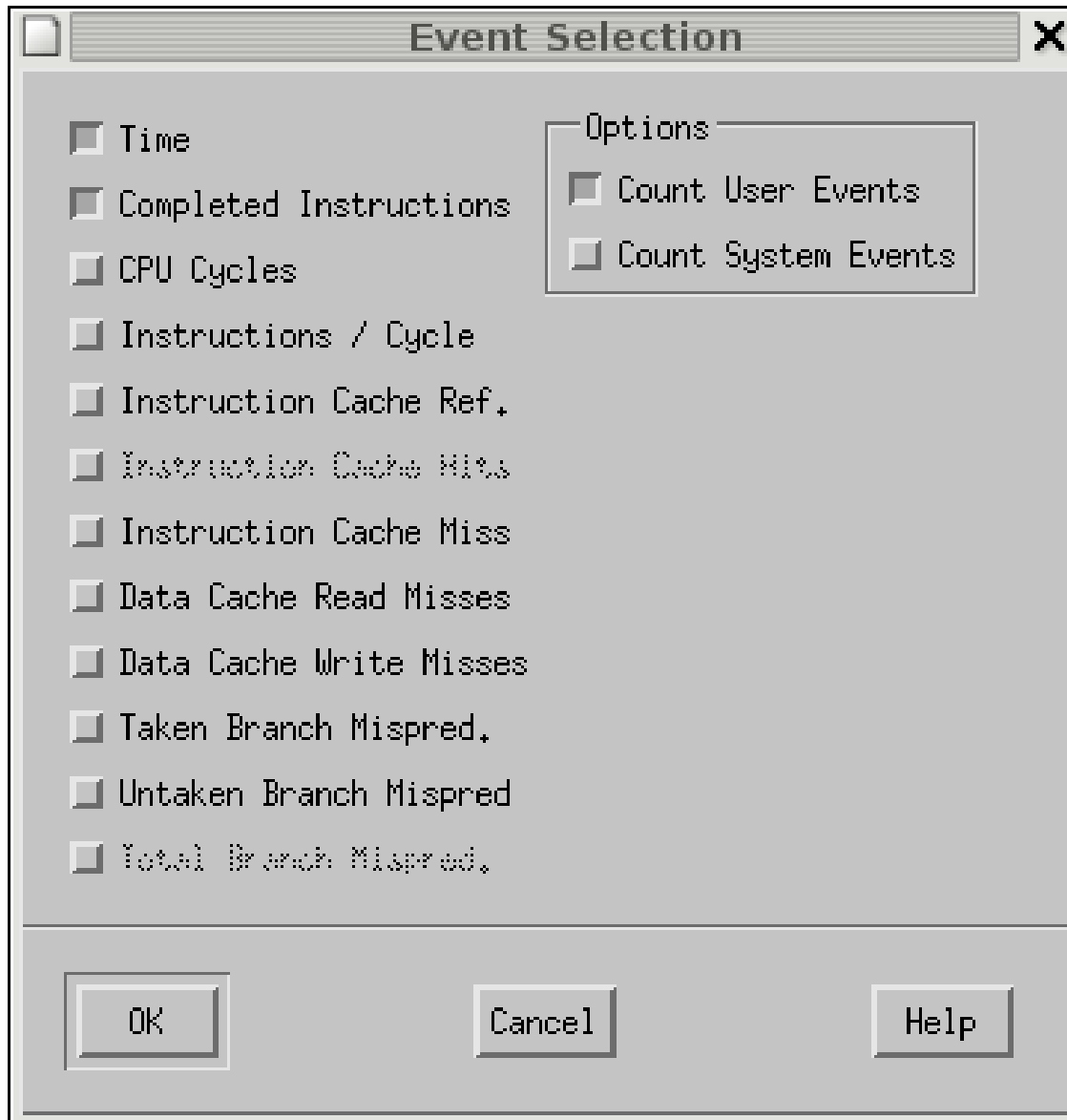
No.	State	Start Time (msec)	Duration (msec)	Cum. Duration (msec)	Transition in Routine
6	ready	5.806	0.758	1.328	Philosopher::main
7	running	6.564	0.006	1.334	Philosopher::main
8	blocked	6.570	0.041	1.375	uSemaphore::P
9	ready	6.611	0.829	2.204	uSemaphore::P
10	running	7.440	0.004	2.208	uSemaphore::P
11	ready	7.444	0.030	2.238	Philosopher::main
12	running	7.474	0.008	2.246	Philosopher::main
13	ready	7.482	0.276	2.522	Philosopher::main
14	running	7.758	0.004	2.526	Philosopher::main
15	blocked	7.762	0.040	2.566	uSemaphore::P
16	ready	7.802	0.014	2.580	uSemaphore::P
17	running	7.816	0.003	2.583	uSemaphore::P
18	ready	7.819	0.015	2.598	Philosopher::main
19	running	7.834	0.007	2.605	Philosopher::main
20	ready	7.841	0.013	2.618	Philosopher::main
21	running	7.854	0.004	2.622	Philosopher::main
22	blocked	7.858	0.038	2.660	uSemaphore::P



Call Graph Metrics

- Highly accurate exact or statistical call graph
- **exact**: profiling data collected at each routine call/exit/block/unblock
 - greater accuracy but greater overhead
 - dynamically create calling-context tree to maintain profiling data (per task, per coroutine)
- **statistical**: profiling data collected at specific time intervals (sampling periods)
 - lower overhead but lower accuracy
 - take complete call-stack per sample and dynamically create an approximate calling-context tree (per task)
- No gprof fallacy, and call-graph is connected
- Measure hardware events or time
- Navigating a large call graph is important

Exact Call Graph



Routine Call Graph, Task/Coroutine Selection				
Close				
Task/Coroutine Name (ID)	Time (seconds)		Instr. Count	
	Total Exec.	Total Block	Total Exec.	Total Block
uBootTask (0x112070)	166u	0	21.89k	0
uSystemTask (0x131a20)	0	0	0	0
uMain (0xffbef640)	1.984m	16.02m	324.2k	4.604M
T1 (0xcb7488)	5.772m	0	1.674M	0
T2 (0xcb72c0)	6.399m	0	1.813M	0

uInorderTreeGen (0xcb77	1.387m	0	457.5k	0
uPreorderTreeGen (0xcb7l	1.372m	0	457.1k	0
uPostorderTreeGen (0xcbi	4.726m	0	2.743M	0
uPostorderTreeGen (0xcbi	728u	0	272.1k	0

Routine Call Graph: Task T1 (0xcb7488)

Time - Seconds: 6.38m

Calls + Continues	% of Total	Self	% of Total	Desc.	Block	Routine
7: 20+0		66u		0	0	Treeable::Treeable
8: 20+0		122u		66u	0	mynode::mynode
9: 3+0		13u		0	0	Tree<mynode>::top
10: 3+0		20u		13u	0	uTreeIter<mynode>::i
11: 3+0		23u		33u	0	uTreeIter<mynode>::o
12: 1+0		315u		7,286m	0	fred::main
13: 41+210		459u		134u	0	uInorderTreeGen<myno
14: 1+20		15u		593u	0	uTreeIter<mynode>::m
15: 20+0		68u		0	0	TFriend::left
16: 20+0		66u		0	0	TFriend::right

call-graph routines

Callers of uTreeIter<mynode>::init (Time - Seconds)

Calls + Continues	Self	Desc.	Block	Routine
11: 3+0	20u	13u	0	uTreeIter<mynode>::over

callers of selected routine

Callees of uTreeIter<mynode>::init (Time - Seconds)

Calls + Continues	Self	Desc.	Block	Routine
9: 3+0	13u	0	0	Tree<mynode>::top

callees of selected routine

Callees Visited

uTreeIter<mynode>::init

current call-graph path visited

Call Cycles

Cycle 1: Tree<mynode>::insertNode -> Tree<mynode>::insertNode
 Cycle 2: uInorderTreeGen<mynode>::findNextNode -> uInorderTreeGen<mynode>::findNextNode

call-graph cycles

Coroutine Selection

uInorderTreeGen (0xcb7770)
 uPreorderTreeGen (0xcb76e0)
 uPostorderTreeGen (0xcb7650)

coroutines executed

Routine Call Graph: Task T1 (0xcb7488)

Close Options

Time - Seconds

- Histogram
- File Info
- Events
- Complete Call Graph

Time - Seconds	Self	% of Total	Desc.	Block	Routine
7: 0	0			0	Treeable::Treeable
8: 66u	0			0	mynode::mynode
9: 3+0	13u			0	Tree<mynode>::top
10: 3+0	20u	13u		0	uTreeIter<mynode>::i
11: 3+0	23u	33u		0	uTreeIter<mynode>::o
12: 1+0	315u	7,286m		0	fred::main
13: 41+210	459u	134u		0	uInorderTreeGen<myno
14: 1+20	15u	593u		0	uTreeIter<mynode>::m
15: 20+0	68u	0		0	TFriend::left
16: 20+0	66u	0		0	TFriend::right

Callers of uTreeIter<mynode>::init (Time - Seconds)

Time - Seconds	Calls + Continues	Self	Desc.	Block	Routine
11: 3+0	20u	13u		0	uTreeIter<mynode>::over

Callees of uTreeIter<mynode>::init (Time - Seconds)

Time - Seconds	Calls + Continues	Self	Desc.	Block	Routine
9: 3+0	13u	0		0	Tree<mynode>::top

Callees Visited

uTreeIter<mynode>::init

Call Cycles

Cycle 1: Tree<mynode>::insertNode -> Tree<mynode>::insertNode
 Cycle 2: uInorderTreeGen<mynode>::findNextNode -> uInorderTreeGen<mynode>::findNextNode

Coroutine Selection

uInorderTreeGen (0xcb7770)
 uPreorderTreeGen (0xcb76e0)
 uPostorderTreeGen (0xcb7650)

Call Graph								
Weight		Time		Completed Instructions		Routine		
		Time	Completed Instructions	Block	Self	Desc.	Block	
	20+0			0	67.61k	1,233M	0	Tree<mynode>::insert
	190+0	2.427m	1,639m	0	609.1k	559.4k	0	Tree<mynode>::insertNode
23	210+0	2.695m	1,836m	0	676.7k	623.6k	0	Tree<mynode>::insertNode
	190+0	615u	0	0	199.1k	0	0	operator<
	190+0	614u	0	0	213.2k	0	0	operator>
	190+0	607u	0	0	211.4k	0	0	TFriend::right
	190+0	2.427m	1,639m	0	609.1k	559.4k	0	Tree<mynode>::insertNode

	190+0	607u	0	0	211.4k	0	0	Tree<mynode>::insertNode
20,81	190+0	607u	0	0	211.4k	0	0	TFriend::right

	190+0	615u	0	0	199.1k	0	0	Tree<mynode>::insertNode
20,81	190+0	615u	0	0	199.1k	0	0	operator<

	190+0	614u	0	0	213.2k	0	0	Tree<mynode>::insertNode
20,81	190+0	614u	0	0	213.2k	0	0	operator>

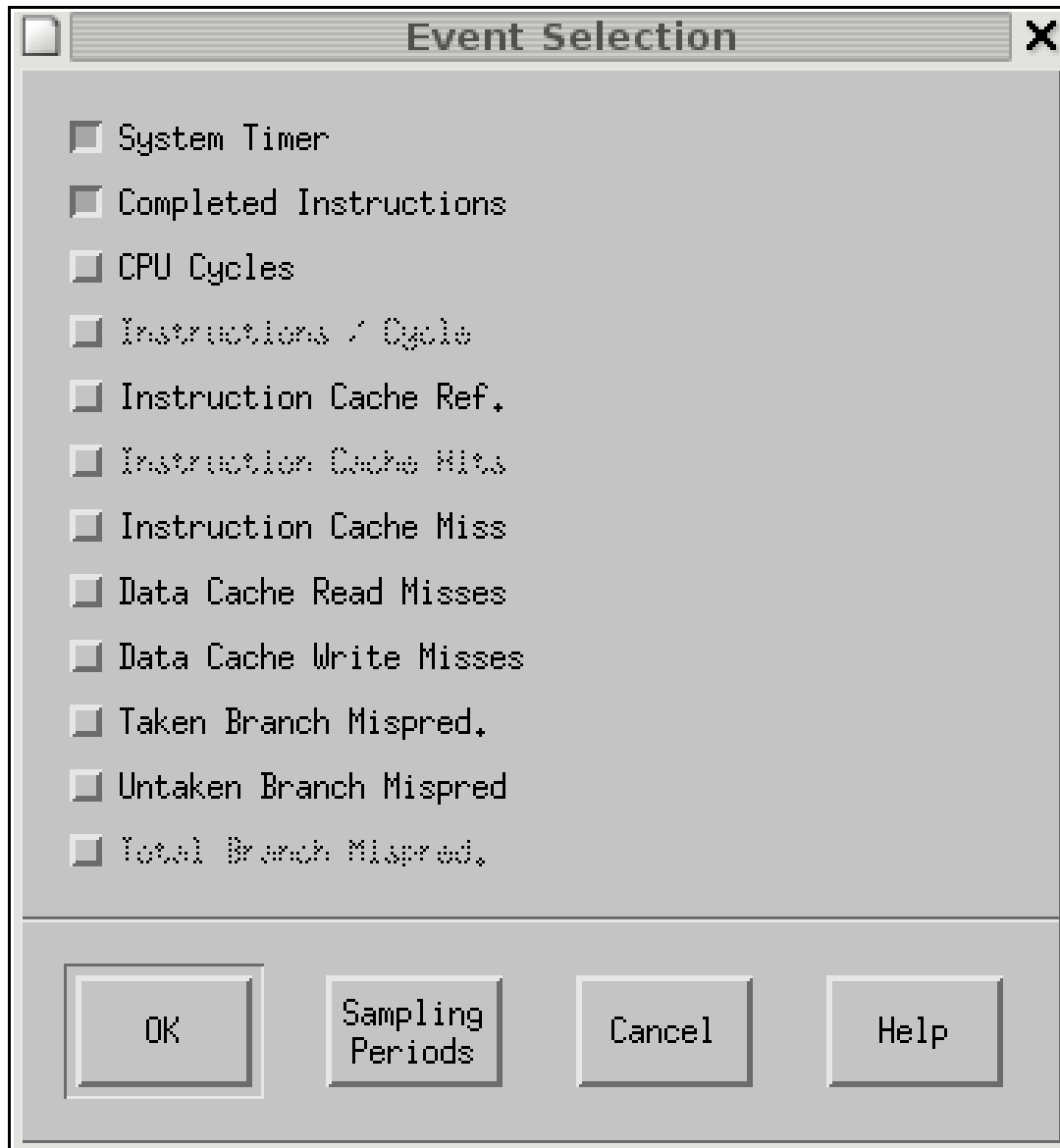
	63+0	564u	1,829m	0	198.4k	530.2k	0	fred::main
6,9	63+0	564u	1,829m	0	198.4k	530.2k	0	uTreeIter<mynode>::operator>>

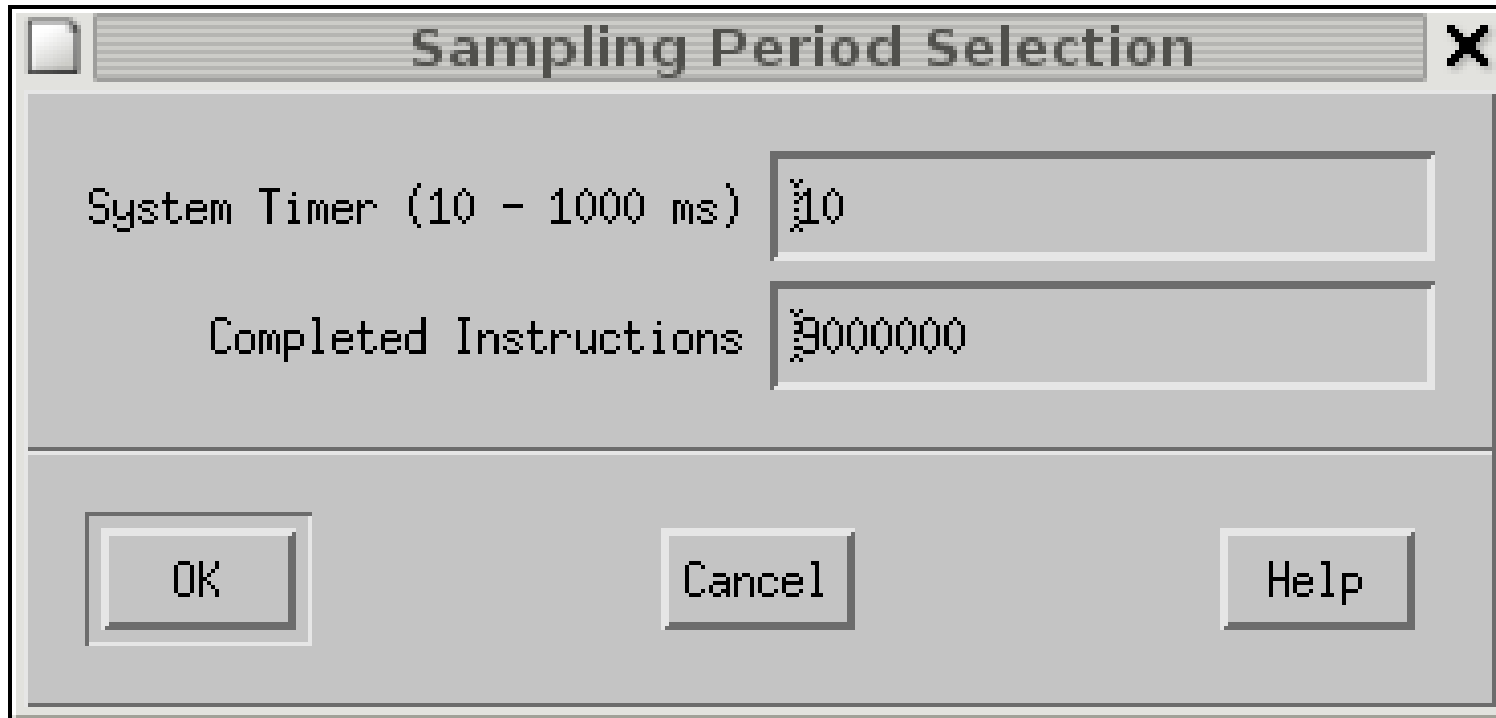
	20+0	118u	4,531m	0	33.72k	1.3M	0	fred::main
2,191	20+0	118u	4,531m	0	33.72k	1.3M	0	Tree<mynode>::insert
	20+0	268u	4,263m	0	67.61k	1,233M	0	Tree<mynode>::insertNode

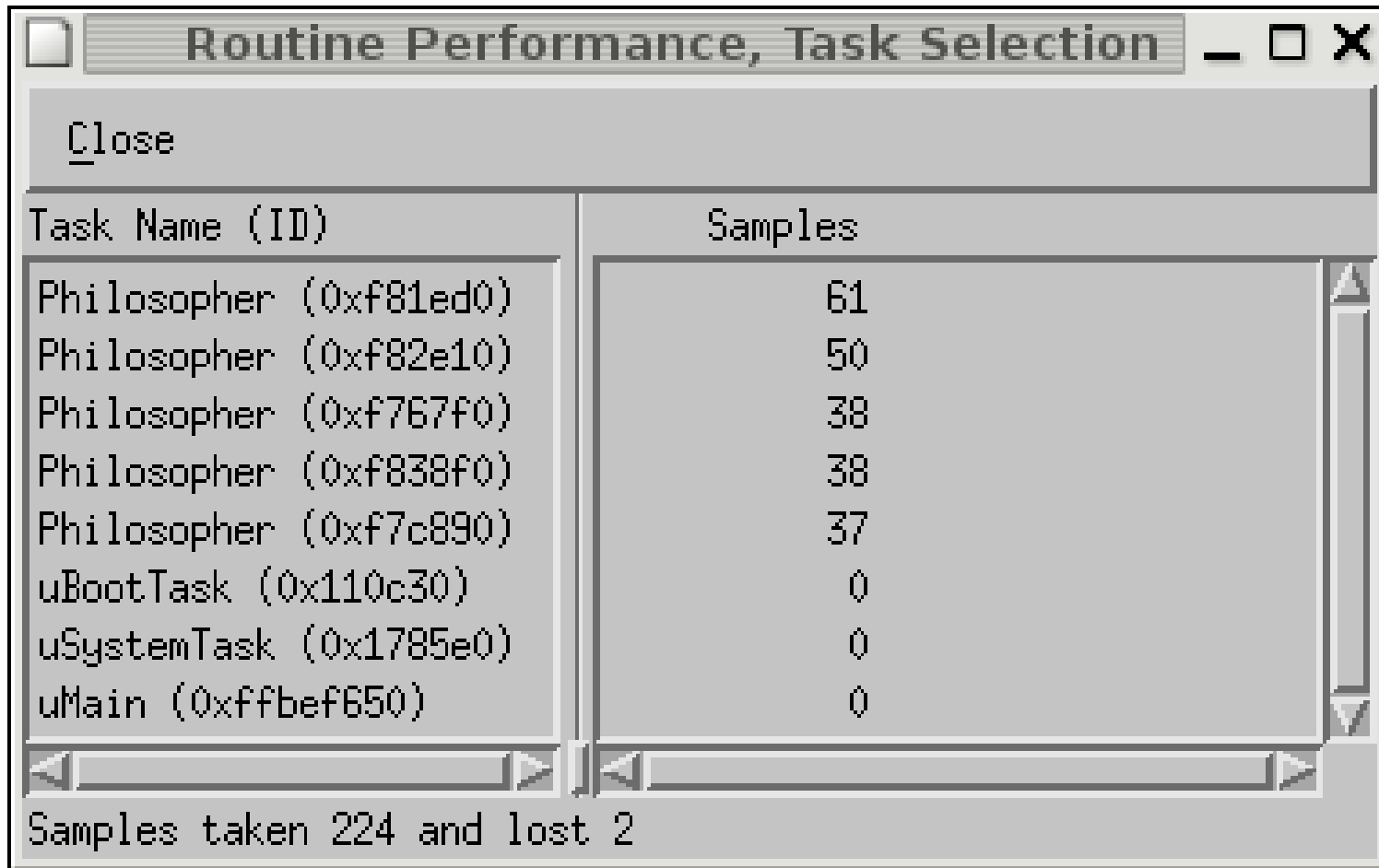
	20+0	66u	0	0	21.45k	0	0	mynode::mynode
2,191	20+0	66u	0	0	21.45k	0	0	Treeable::Treeable

	20+0	122u	66u	0	33.75k	21.45k	0	fred::main
2,191	20+0	122u	66u	0	33.75k	21.45k	0	mynode::mynode
	20+0	66u	0	0	21.45k	0	0	Treeable::Treeable

Statistical Call Graph







The screenshot shows a window titled "Routine Performance, Task Selection" with a "Close" button. The window contains a table with two columns: "Task Name (ID)" and "Samples". The table lists several tasks, including five instances of "Philosopher" with varying sample counts, and three other tasks: "uBootTestTask", "uSystemTask", and "uMain", all with zero samples. The window also displays the text "Samples taken 224 and lost 2" at the bottom.

Task Name (ID)	Samples
Philosopher (0xf81ed0)	61
Philosopher (0xf82e10)	50
Philosopher (0xf767f0)	38
Philosopher (0xf838f0)	38
Philosopher (0xf7c890)	37
uBootTestTask (0x110c30)	0
uSystemTask (0x1785e0)	0
uMain (0xffbef650)	0

Samples taken 224 and lost 2

Routine Performance : Task Philosopher (0xf81ed0)

Close Options

System Timer (sample period 10 ms) - Samples: 38

Self	% of Max Self	% of Total	Self+Desc.	% of Total	Routine
6: 2			3		uBaseTask::yield
7: 2			7		uCondition::wait
8: 2			19		uSemaphore::P
9: 2			10		uSemaphore::V
10: 2			2		uSerial::leave
11: 1			1		Table::LeftOf
12: 1			2		Table::TestBeside
13: 1			14		Table::putdown
14: 1			2		uProcessorKernel::schedule
15: 1			1		uProcessorKernel::scheduleInter

selected routine (arrow pointing to Table::TestBeside)

call-graph routines

Callers of Table::TestBeside (System Timer - Samples)

Self	Descendents	Routine
19: 0	1	Table::pickup
13: 1	0	Table::putdown

callers of selected routine

Callees of Table::TestBeside (System Timer - Samples)

Self	Descendents	Routine
11: 1	0	Table::LeftOf
9: 0	0	uSemaphore::V

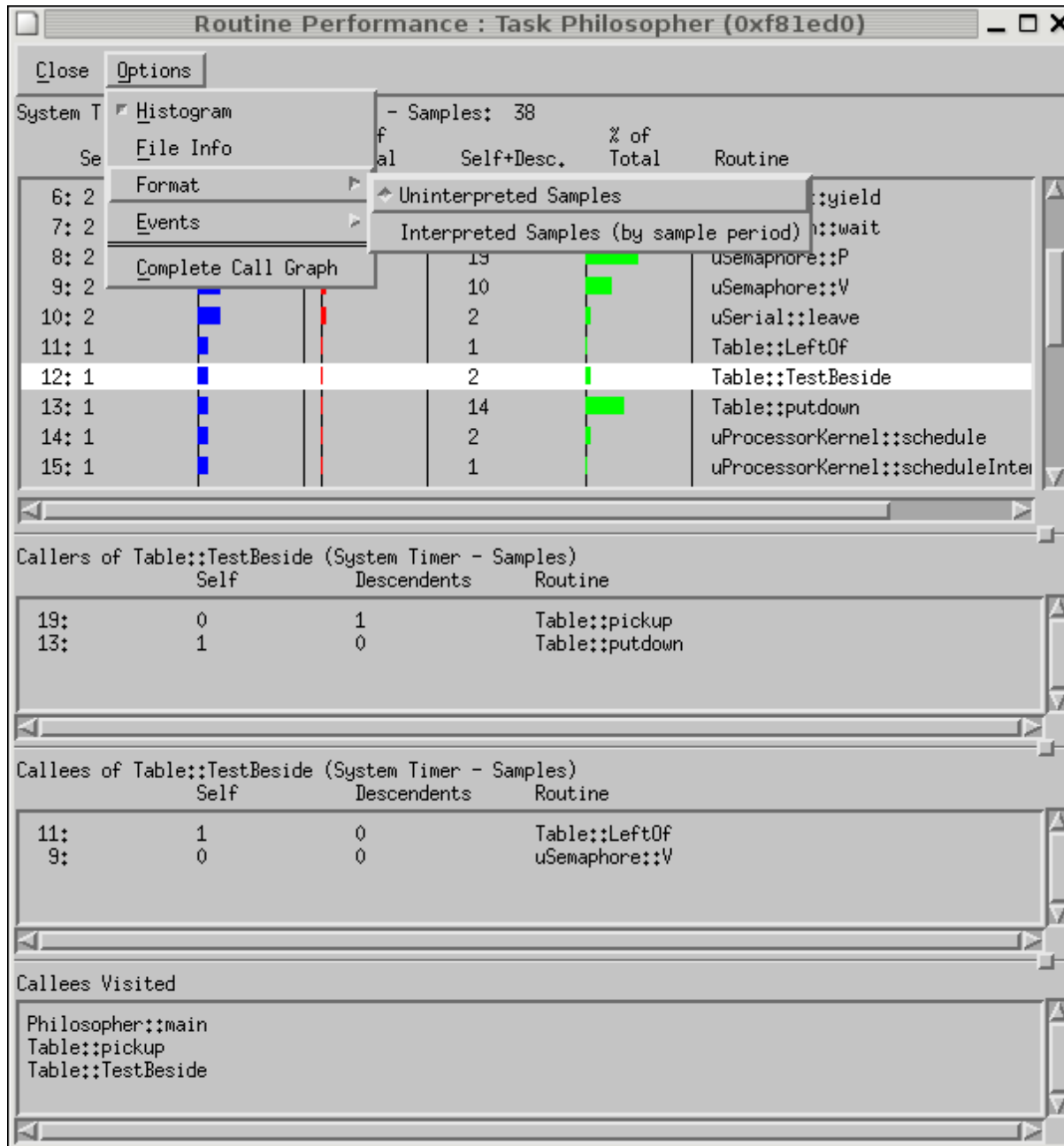
callees of selected routine

Callees Visited

```

Philosopher::main
Table::pickup
Table::TestBeside
    
```

current call-graph path visited



Profiler Performance

- μ Profiler Execution state transition
 - 70% probe effect on each state transition (worst case scenario)
- Call Graph
 - fixed call depth (8) with increasing number of calls at each depth (wide tree)
 - each leaf node is called 30,000 times
 - sample every **.7ms** using **CPU cycle-counter**

Statistical			S/E	Exact	
μ Profiler statistical	Sun Studio	HP Caliper	gprof	μ Profiler exact	Intel VTune
2.92%	4.47%	0.60%	791%	21,541%	19,273%

Conclusions

- scalability for exec-state transition and call-graph metrics
- profiling new forms of control flow, i.e., coroutines, at high-level
- competitive with industry profilers in both features and performance
- like to share μ Profiler technology with HP, Intel, SGI

Future Work

- saving & loading metric data
- real time (on-the-fly) visualization
- object-based profiling (including monitors)
- sorting / searching
- automatic calibration and randomization of sampling periods
- issues with statistical cycles (eliminate partial cycles)
- estimate routine-call counts
- release μ Profiler