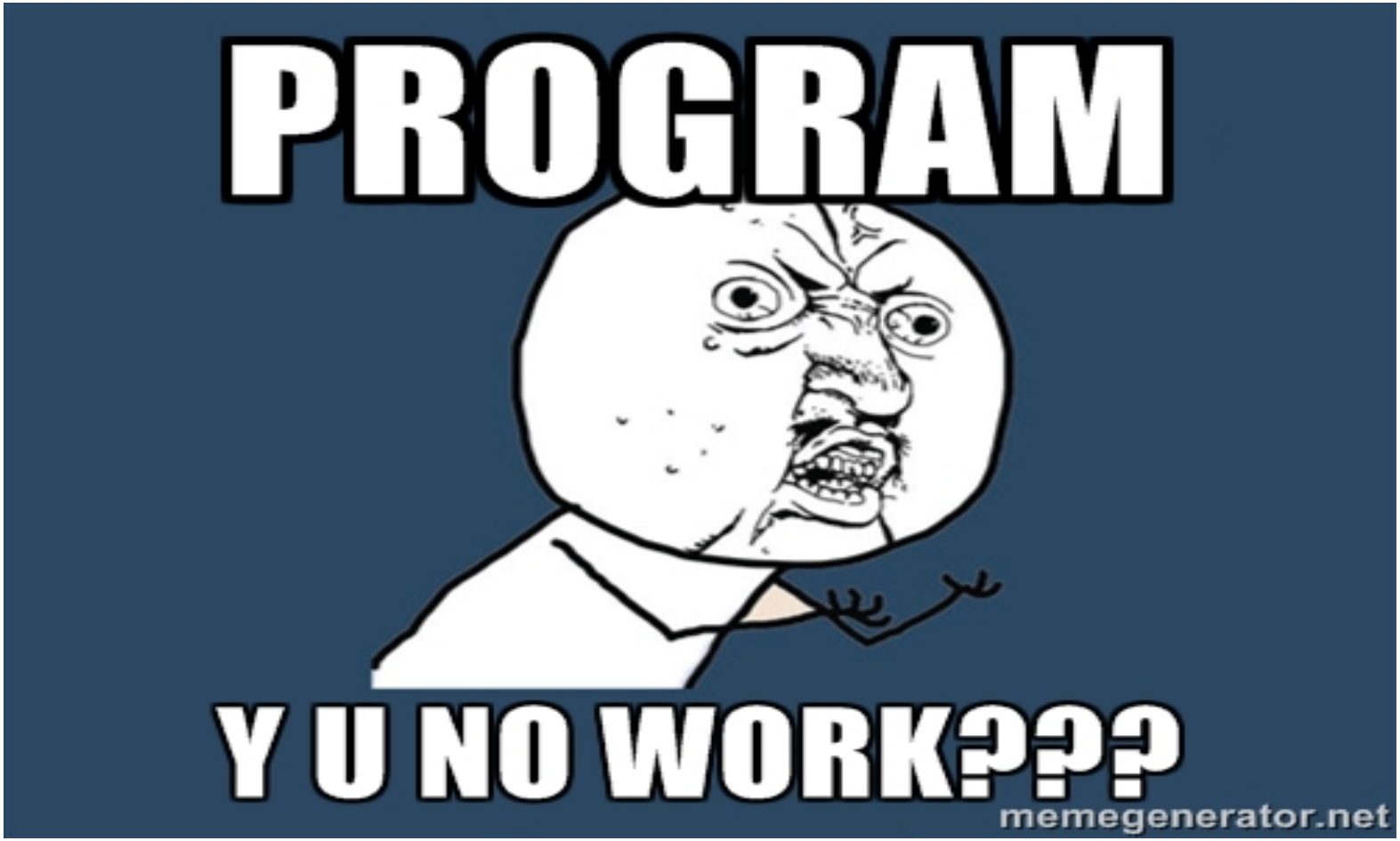
Software Execution Analysis

Philipp Schäfer

February 5, 2014

Outline

Outline


PROGRAM
Y U NO WORK???
memegenerator.net

Domains of Problem Diagnosis

- **Sporadic domain**
  - problem source not known
  - occurs in an asynchronous and random manner
  - system faults
- **Temporal domain**
  - problem scope narrowed but not reproducible
- **Logical domain**
  - error reproducible
  - exact sequence of conditions or events that triggers the error was found

Definition:

In software engineering, **profiling** is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls.

- *Flat profilers* compute the average call times, from the calls, and do not break down the call times based on the callee or the context.
- *Call graph profilers* show the call times, and frequencies of the functions, and also the call-chains involved based on the callee.
- *Input-sensitive profilers* generate charts that characterize how an application's performance scales as a function of its input.

Definition:

In software engineering, **tracing** is a specialized use of logging to record information about a program's execution.

- *printf()* tracks a program's progress
- Unix *top* can monitor task creation and track resources
- code coverage and application profiling by compiler-driven instrumentation techniques
- kernel-level instrumentation techniques for accurate timing and process/thread interaction traces

Use Cases

Software Profiling	Event Tracing	Full Tracing
• logical domain • error is definitely caused by application • get insights on application timings and performance	• all domains (mainly sporadic/temporal) • only specific events are logged (threads, functions, IRQs etc)	• all domains (mainly sporadic) • additional hardware required • records/logs nearly everything

Profiling Characteristics	Tracing Constraints
• well discovered and easy to use • no additional hardware needed • contain sets of performance events and timing for execution • in general, no chronological order	• no significant impact on system behavior • exact chronological order of events with fine granular timestamps • handle and log an enormous amount of data (challenging on systems with little memory) • scalable for multi threaded tracing

Outline

- ships with most Linux distributions
- determine which parts of a program are taking the most of execution time
- compile with *-pg*
- link with *-pg*
- execute program $\Rightarrow$ should generate *gmon.out*
- generate profile via *gprof options [executable-file [profile-data-files...]] [> outfile]*

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.50      0.06     0.06                             ftuTransformation::xForm(...) const
 12.50      0.08     0.02  1050226     0.00     0.00  QPointF::QPointF()
  6.25      0.09     0.01  1062192     0.00     0.00  operator new(unsigned long, void*)
  6.25      0.10     0.01   410920     0.00     0.00  bool qMapLessThanKey<QChar>(...)
  6.25      0.12     0.01    16080     0.00     0.00  QBitArray::setBit(int,bool)
  6.25      0.13     0.01      520     0.02     0.02  bubblePlottable::drawQuartileBox(...) const
  6.25      0.14     0.01        1    10.00    10.00  ftuGui::qt_static_metacall(...)
  3.13      0.15     0.01   137994     0.00     0.00  QBasicAtomicInt::operator!=(int) const
```

Listing 1: GNU profiler flat profile

```
granularity: each sample hit covers 2 byte(s) for 6.25% of 0.16 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     37.5    0.06    0.00                 ftuTransformation::xForm(...) const [1]
                0.00    0.00 1839900/1839900     QVector<QwtInterval>::size() const [353]
-----------------------------------------------
                                                 <spontaneous>
[2]     20.1    0.00    0.03                 ftuCommunicate::qt_static_metacall(...) [2]
                0.00    0.03       1/1           ftuCommunicate::stopReadOut() [3]
                0.00    0.00     390/390         ftuCommunicate::socketReadyReadout() [186]
                0.00    0.00      40/40          ftuCommunicate::addCurve(QChar) [219]
                0.00    0.00       1/1           ftuCommunicate::startReadOut() [346]
                0.00    0.00     432/431610      qt_noop() [355]
-----------------------------------------------
                0.00    0.03       1/1           ftuCommunicate::qt_static_metacall(...) [2]
[3]     20.0    0.00    0.03       1         ftuCommunicate::stopReadOut() [3]
                0.00    0.03       1/1           ftuCommunicate::addDataToPlot(QByteArray*) [4]
                0.00    0.00      40/40          ftuPlotCurve::appendPoint(double, double) [70]
                0.00    0.00       1/1           statisticalPlot::updatePlot() [206]
                0.00    0.00       1/1           bubblePlot::updatePlot() [225]
                0.00    0.00      40/118         QVector<double>::last() [240]
                0.00    0.00       1/44          ftuLog::log(QString const&) [266]
                0.00    0.00      80/80          ftuPlotCurve::getYData() const [916]
```

Listing 2: GNU profiler call graph

- include heap profiler, heap checker, CPU profiler and malloc/free implementation (TCMalloc)
- heap checker: detect memory leaks, multiple modes of heap leak checking
- heap profiler: locate memory leaks, locate unnecessary memory allocations
- CPU profiler: like gprof but is able to generate a graphical representation of the data

Heap Checker

- dumps a memory usage profile on program start and another one on program exit
- compare profiles to locate leaks
- whole-program checking
  - recommended way
  - significant increase of memory usage
  - can be tweaked with 4 different modes (minimal, normal, strict and draconian)
- partial-program checking
  - analyze only specific parts of program
  - bracket code fragment with creation of *HeapLeapChecker* object and *NoLeaks()* method call

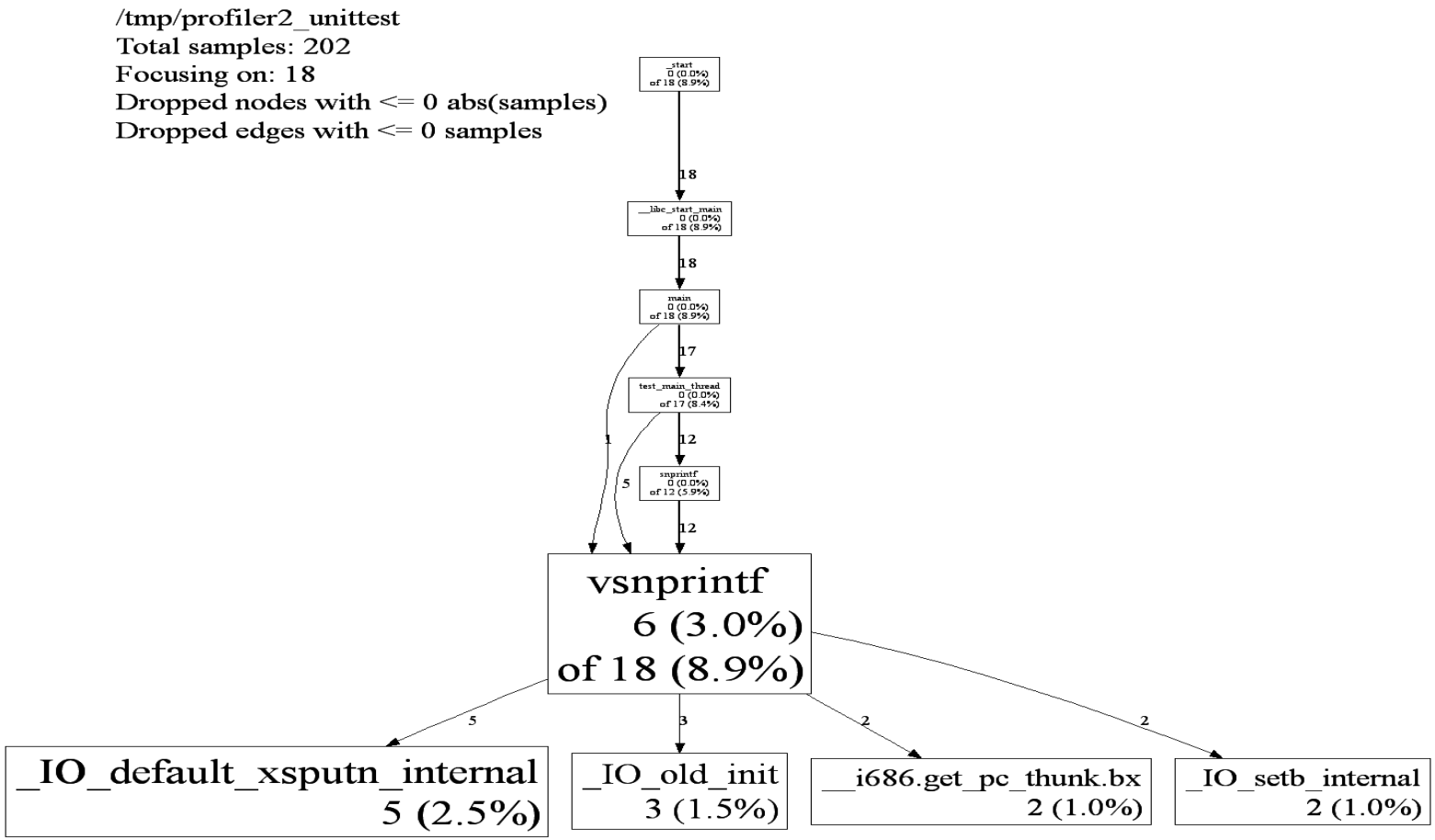


Figure : Google Performance Tool (CPU profiler graphical view)

Outline

Once Again - System Tracing Constraints

- no significant impact on system behavior
- exact chronological order of events with fine granular timestamps
- handle and log an enormous amount of data (challenging on systems with little memory)
- scalable for multi threaded tracing

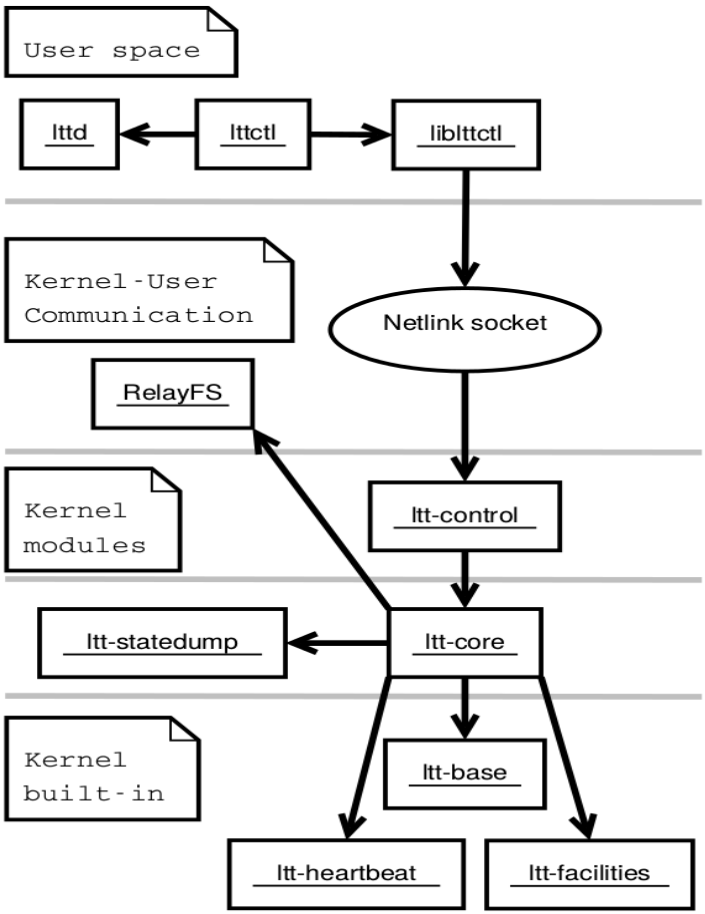


Figure : LTTng control architecture

- 2 user land parts:
  - *lttctl* - command line application which runs in user space
  - *lttd* - user land daemon, waits for trace data and writes it to disk
- *ltt-core* - main module, controls all sub-modules
- RelayFS - provides lockless writing into per-CPU kernel buffers. Can be mmap'ed and read from user space.

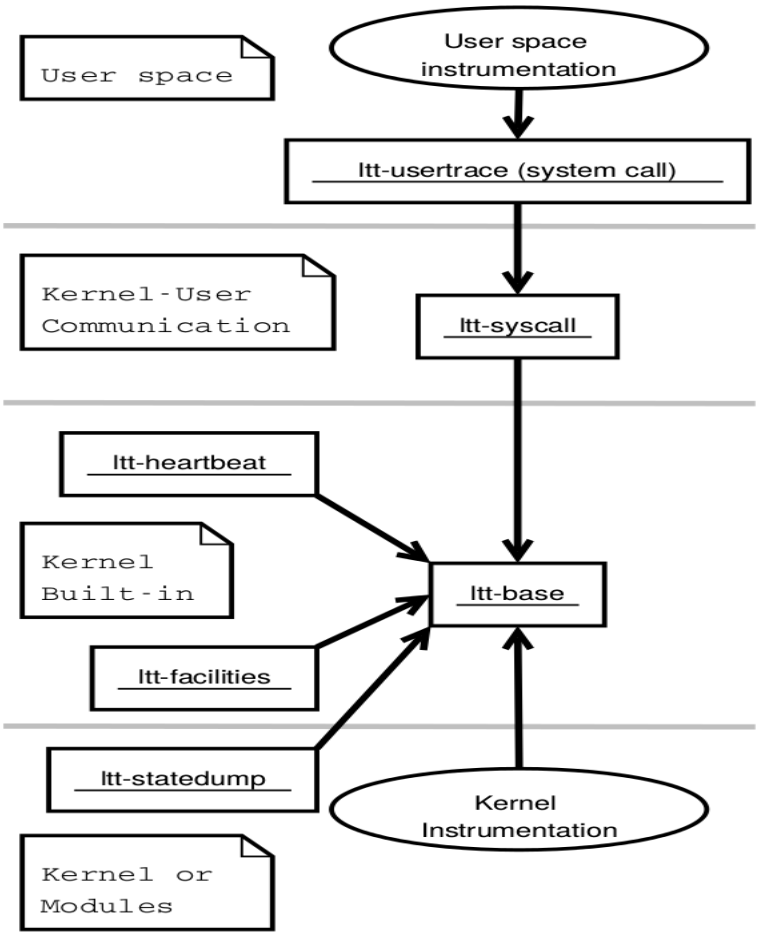


Figure : LTTng tracing

- user-kernel communication via system call
- *ltt-base* gets information from submodules and writes trace to RelayFS buffers
- *ltt-heatbeat* - detect cycle counter overflows
- *ltt-facilities* - lists event types loaded at trace start time
- *ltt-statedump* - generates events to describe kernel state

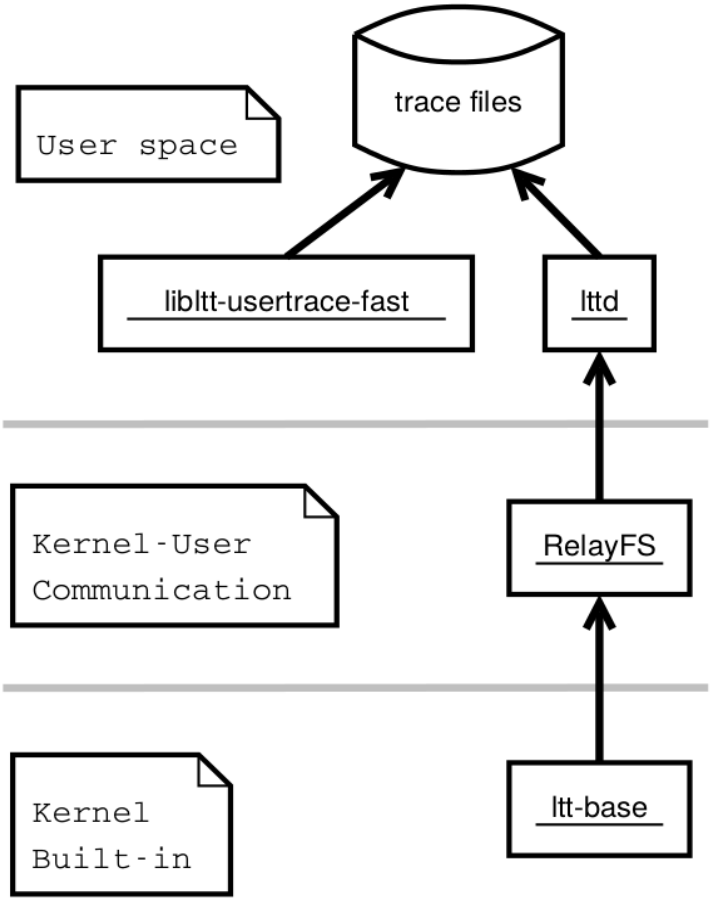


Figure : LTTng data flow

- data is written through *ltt-base* to RelayFS circular buffers
- *lttd* polls on RelayFS channels and writes data to disk
- with *libltt-usertrace-fast*, applications with high data throughput can write traces directly to disk (without system call)

Deploying LTTng on Exotic Embedded Architectures

- LTTng supports: X86 32/64, MIPS, PowerPC 32/64, ARM, S390, Sparc 32/64 and SH64
- porting LTTng to a new architecture:
  - expand instrumentation to include some architecture-specific events
  - kernel_thread_create, syscall_trace, ipc_call, trap_entry, trap_exit, page_fault_entry, page_fault_exit
  - provide an accurate timestamp. Whenever a cycles counter register is available, it should be used.

LTTng Trace Viewers

To oversee the enormous data produced by the tracer, a graphical representation is needed. Some noteworthy viewers:

- Eclipse viewer - plugin
- Linux Trace Toolkit Viewer (LTTV) - standalone
- Percepio Tracealyzer - core support

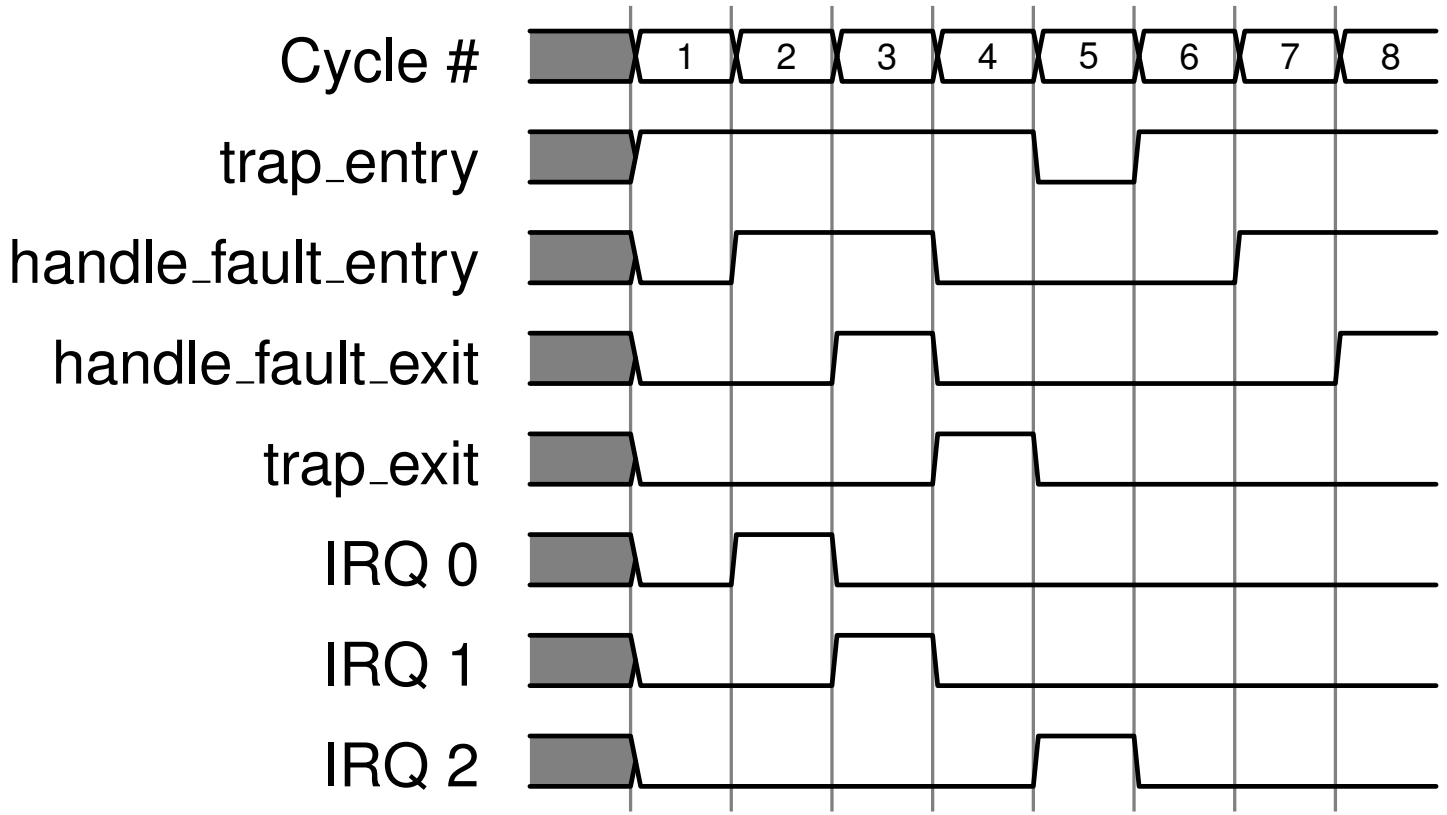


Figure : Example of a waveform like trace representation

- started as ABB developed a control system for industrial robots (IRC 5)
- Trace viewer which supports several traces of different OSes like VxWorks built-in tracing, LTTng traces, FreeRTOS/OpenRTOS, SafeRTOS, rt-kernel and μC/OS-III.
- If no third-party trace, a provided library can be linked (no documentation about functionality)
- worth mentioning because of its several graphical representations

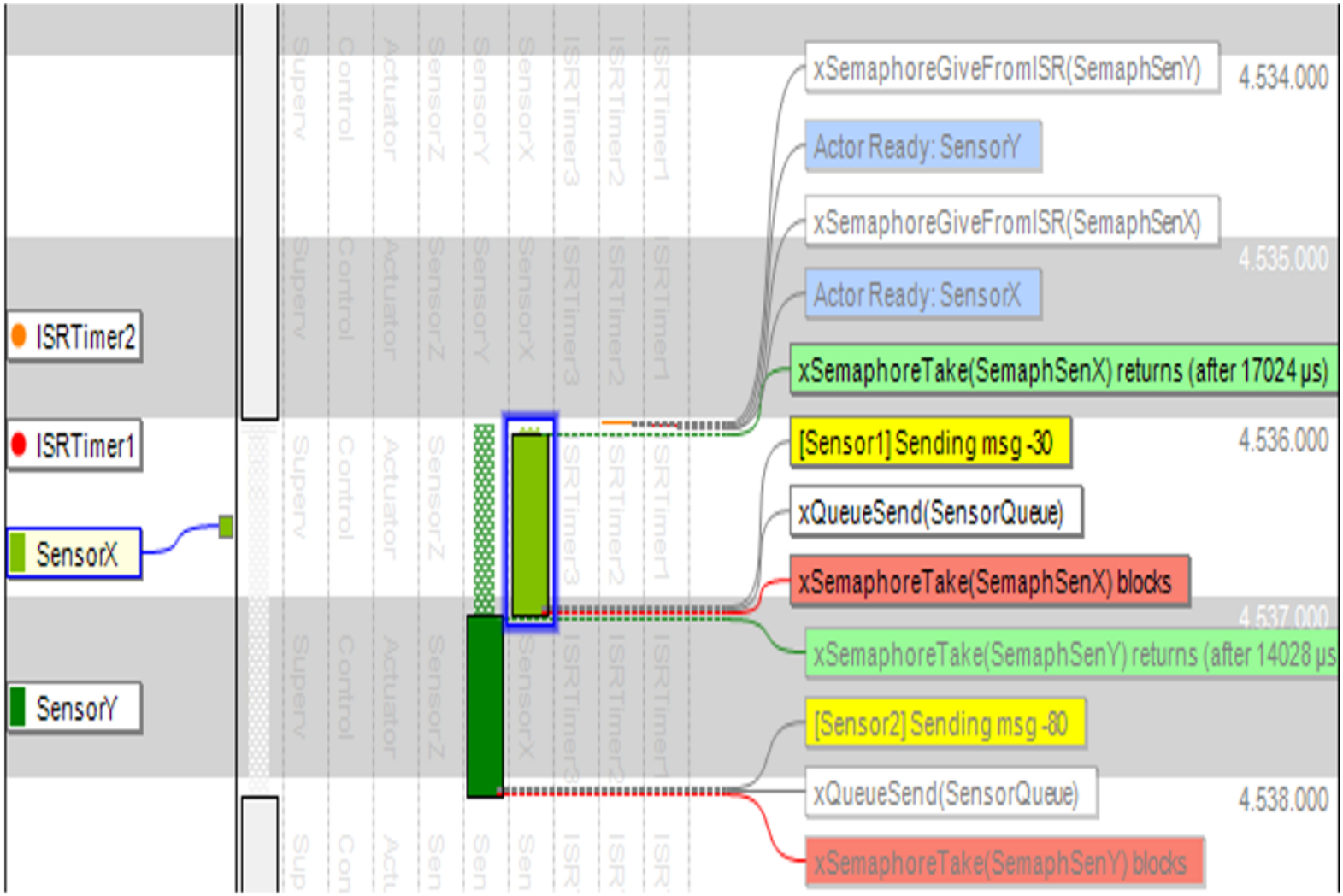


Figure : Percepio Tracealyzer main view

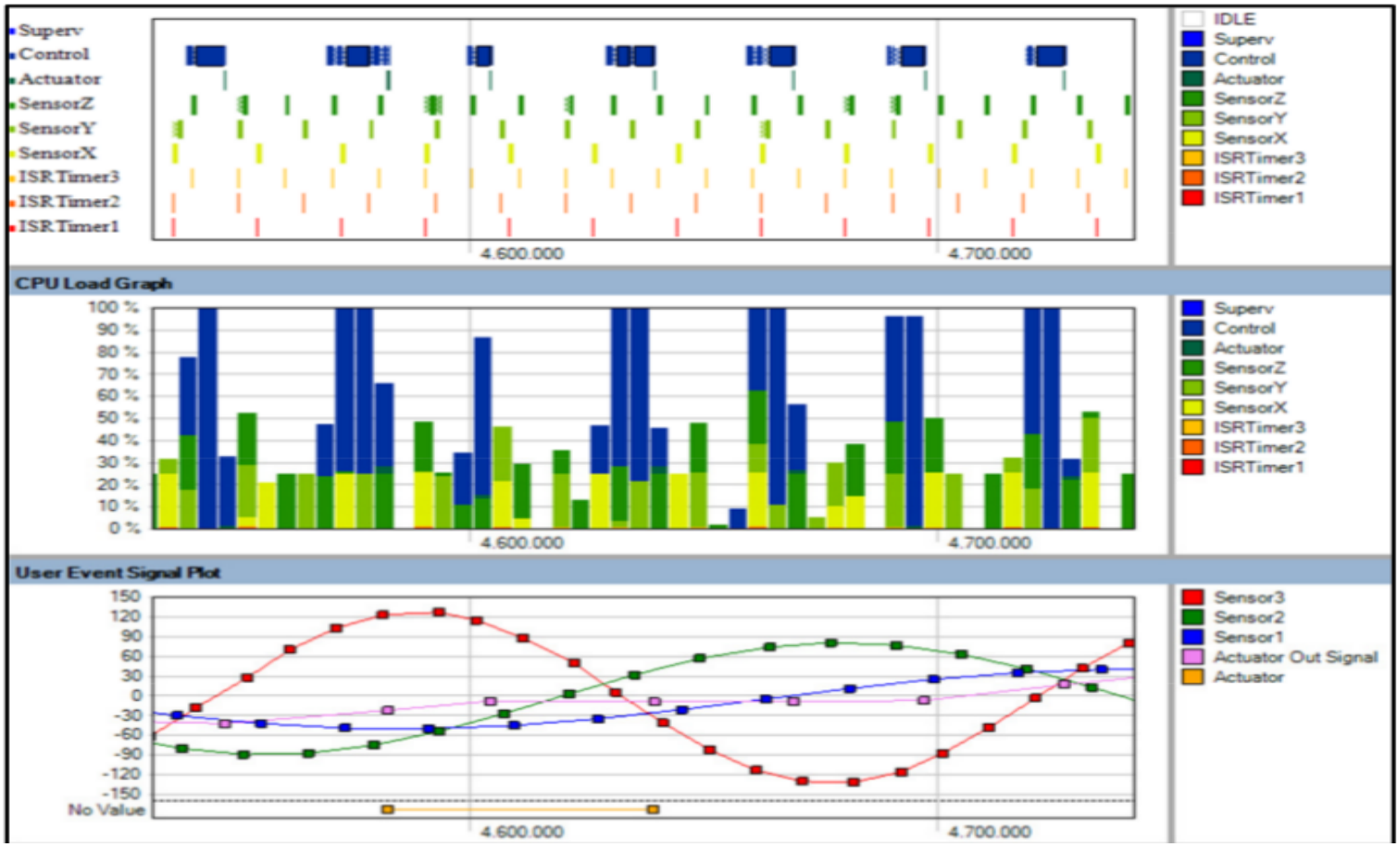


Figure : Percepio Tracealyzer multiple views with synchronized scrolling

- host/target communication via Android Debug Bridge (adb)
- debugging Android in *printf()* like is done via *logcat*
- what if application runs slow or has high CPU usage?
- Android Dalvik Debug Monitor Server (ddms) is used for more detailed debugging it supports
  - port-forwarding services
  - thread and heap information
  - method profiling
  - incoming call, SMS and location data spoofing
  - ...and more
- what about kernel events?

- Android Systrace tool works with Android 4.1+
- needs a kernel with tracing enabled
- in general, it is a python wrapper for *atrace* tracing tool wich is the android extension of *ftrace*
- tracing categories like graphics, input, audio, video, hardware modules, scheduling, activity manager and more
- full trace report is generated on target and read out by host via adb
- Systrace generates a HTML file from atrace output
- Note: LTTng works too on Android

```
# entries-in-buffer/entries-written: 6055/6055   #P:2
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
        DispSync-297   [001] ...1 148716.517760: tracing_mark_write: C|207|VSYNC|0
        DispSync-297   [001] ...1 148716.534472: tracing_mark_write: C|207|VSYNC|1
        DispSync-297   [001] ...1 148716.551139: tracing_mark_write: C|207|VSYNC|0
 ndroid.systemui-1268  [000] ...1 148716.551491: tracing_mark_write: B|1268|performTraversals
 ndroid.systemui-1268  [000] ...1 148716.551553: tracing_mark_write: B|1268|draw
 ndroid.systemui-1268  [000] ...1 148716.551645: tracing_mark_write: B|1268|eglBeginFrame
 ndroid.systemui-1268  [000] ...1 148716.551660: tracing_mark_write: E
 ndroid.systemui-1268  [000] ...1 148716.551691: tracing_mark_write: B|1268|getDisplayList
 ndroid.systemui-1268  [000] ...1 148716.551930: tracing_mark_write: E
 ndroid.systemui-1268  [000] ...1 148716.551975: tracing_mark_write: B|1268|prepareFrame
 ndroid.systemui-1268  [000] ...1 148716.552010: tracing_mark_write: E
 ndroid.systemui-1268  [000] ...1 148716.552041: tracing_mark_write: B|1268|drawDisplayList
 ndroid.systemui-1268  [000] ...1 148716.552159: tracing_mark_write: B|1268|precacheText
```

Listing 3: Android atrace output

Android Systrace

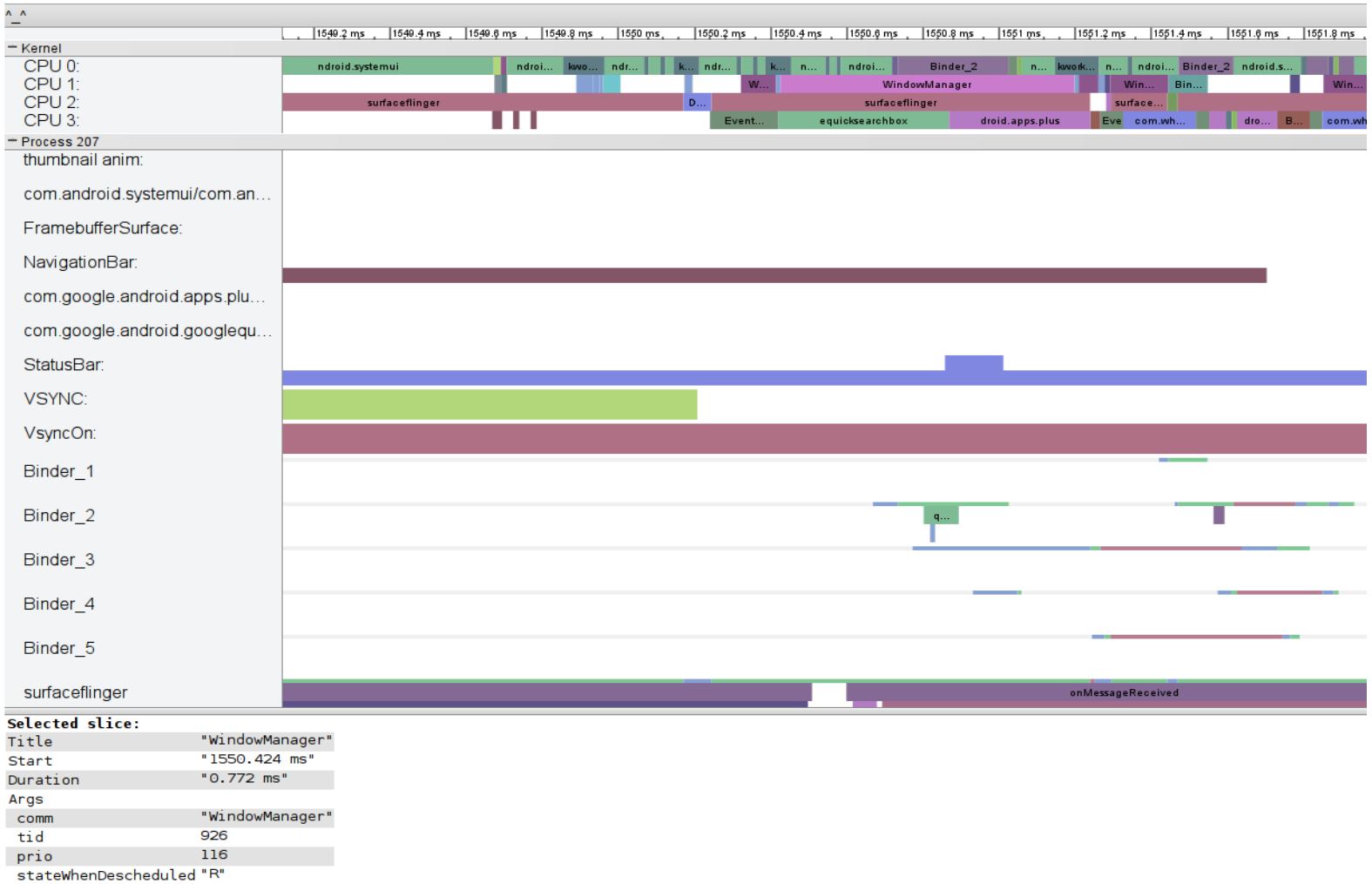


Figure : Android Systrace HTML output

- full custom example RTOS system developed at ZITI Heidelberg
- Digilent Atlys^{TM}Spartan-6 FPGA Development Board
- tracing module requirements
  - high resolution timestamps
  - low overhead
  - low memory consumption

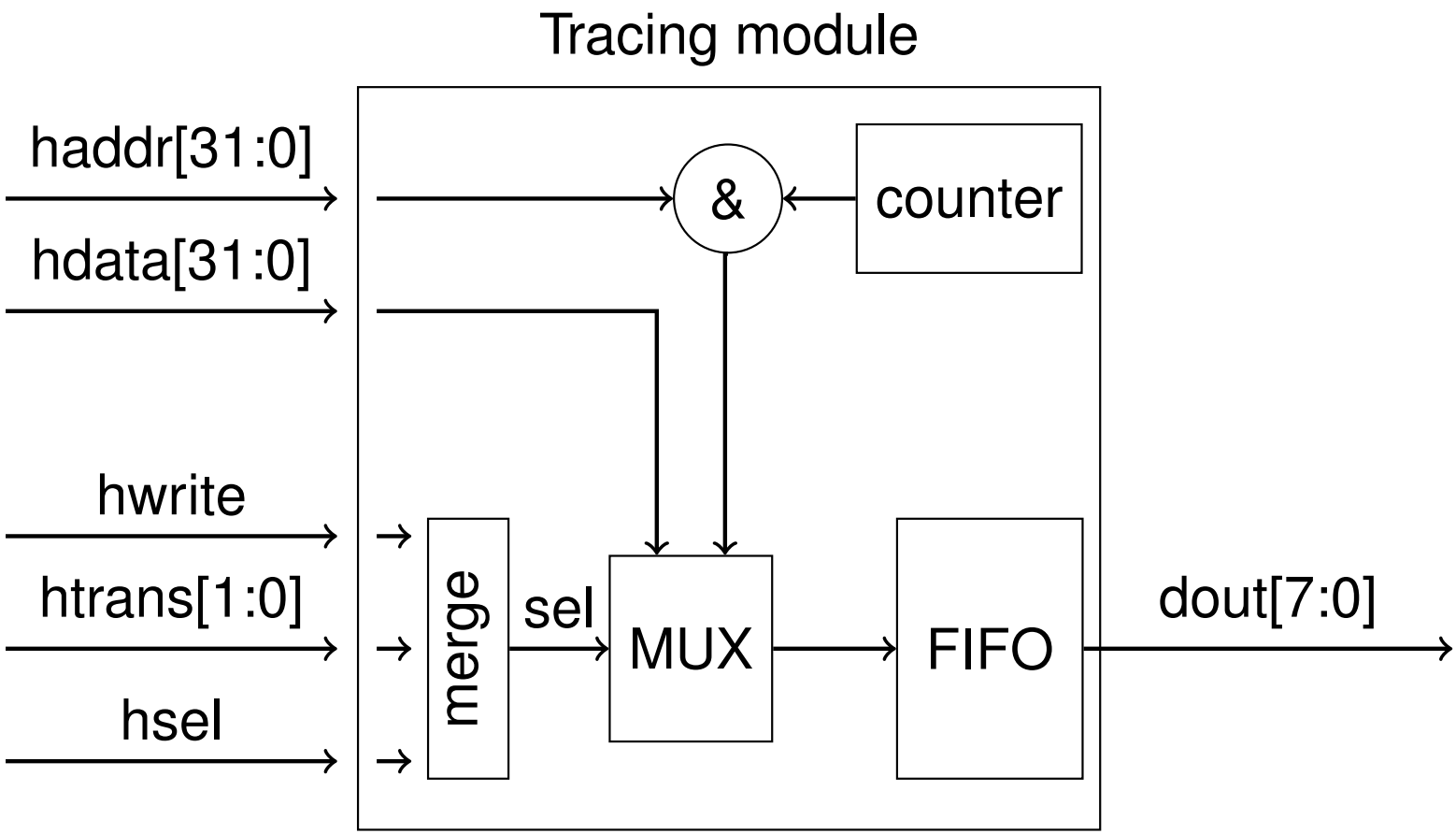


Figure : Example tracing hardware module design

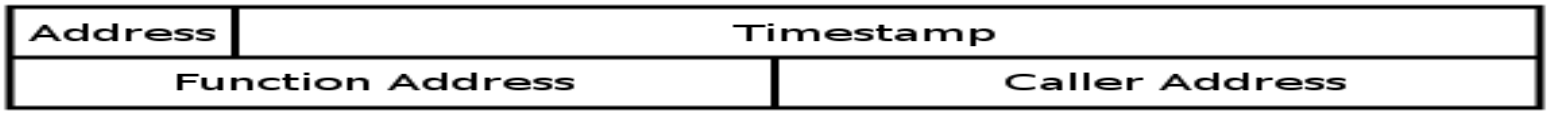


Figure : Bit coding for function entry/exit event

- low memory consumption is achieved by coding every event with only 64 bit
- 2 address bit for 4 event types (function entry/exit, rtos event, misc)
- 30 bit for timestamp
- 32 bit for data depending on event type

- entry/exit events generated with *gcc* and *-finstrument-functions*
- emit calls to *__cyg_profile_func_enter()* and *__cyg_profile_func_exit()*
- defined to push an event to tracing module

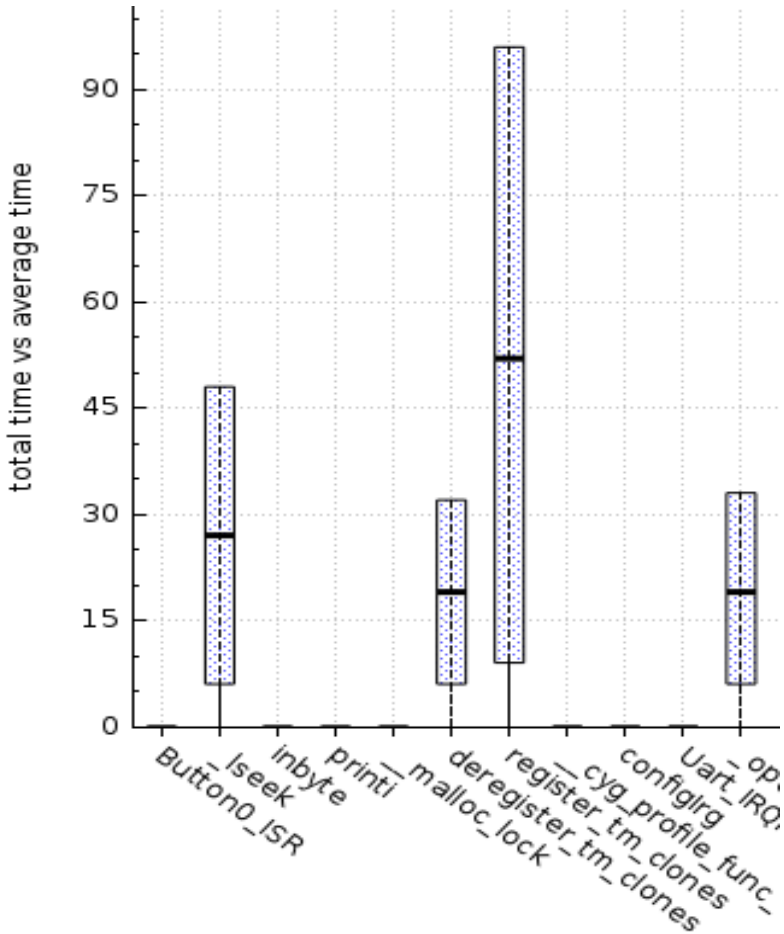


Figure : FTU gui statistical plot

- custom trace view in development called FPGA Trace Utility GUI (FTU gui)
- communication via UDP sockets
- provides 3 different graphical representations
  - waveform
  - minimum/maximum time vs average time in statistical plot
  - number of function calls vs total execution time in bubble blot

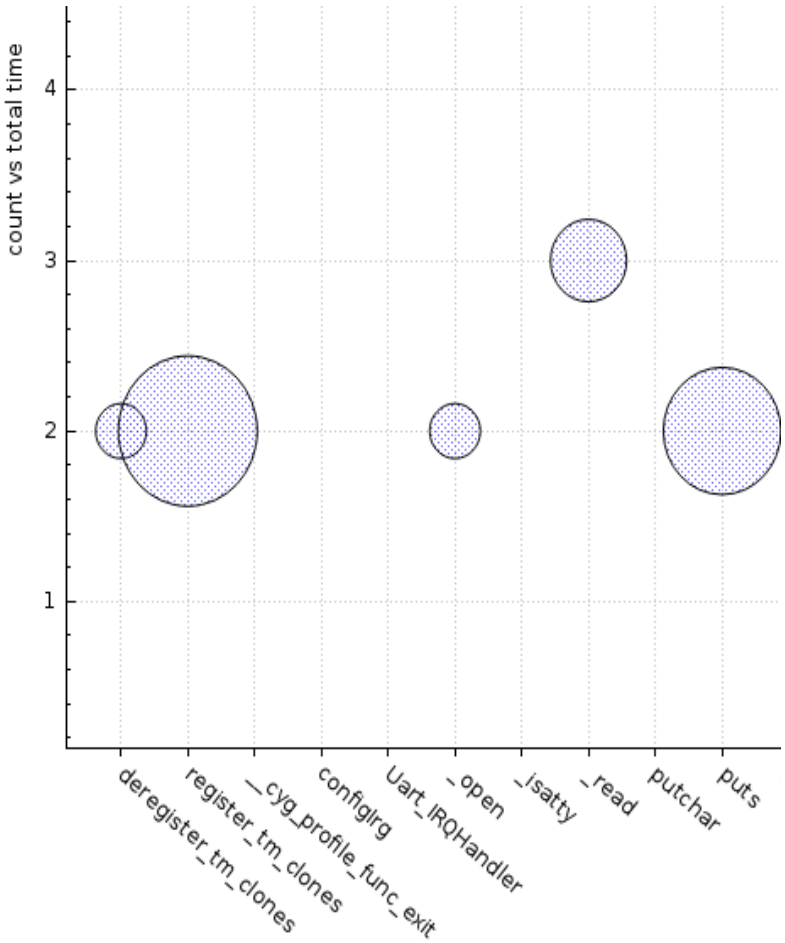


Figure : FTU gui bubble plot

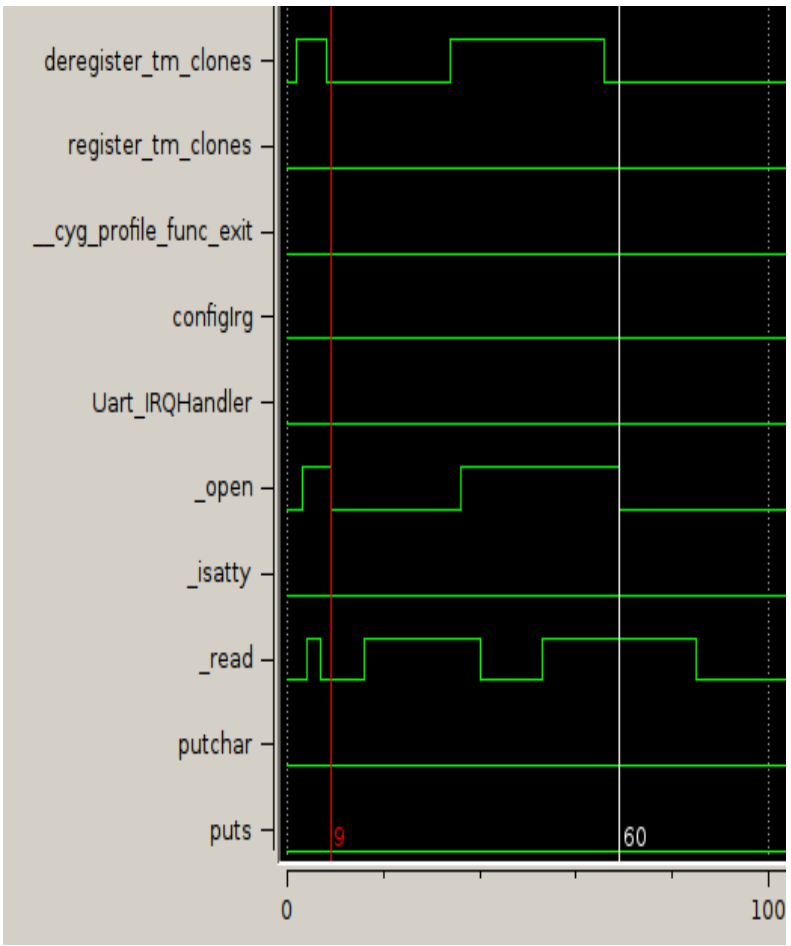


Figure : FTU gui wave plot

Outline

- **Software Profiling**
  - tools are well discovered and easy to use
  - tools like gperf or gprof provide a clean overview on your application
  - provide sets of performance events and timings for execution with no chronological order
  - no additional hardware is needed
- **System Tracing**
  - fine granular timestamps for every event
  - different categories of what to be traced
  - additional hardware may be required
  - OS dependent
  - indispensable for RTOS debugging

Thanks for your attention!

Questions?

For Further Reading I

T. Fletcher
Using System Tracing Tools to Optimize Software Quality and Behavior

M. Desnoyers and M.R. Dagenais
The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux
Ottawa Linux Symposium, 2006

T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore and M.R. Dagenais
relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space
Ottawa Linux Symposium, 2003

For Further Reading II


E.G. Bregnant and D.P.B. Renaux
RTOS Scheduling Analysis using a Trace Toolkit


M. Desnoyers and M.R. Dagenais
Deploying LTTng on Exotic Embedded Architectures


R.W. Wisniewski and B. Rosenburg
Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems


J. Kraft, A. Wall and H. Kienle
Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects

For Further Reading III

J. Fenlason and R. Stallman
GNU gprof - The GNU Profiler
http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html, 1994

Google
GooglePerformanceTools
http://code.google.com/p/gperftools/wiki/GooglePerformanceTools, 2013

Google
Android Debugging
http://developer.android.com/tools/debugging/index.html, 2013