Linux Kernel Validation Tools

Nicholas Mc Guire
Distributed & Embedded Systems Lab
Lanzhou University, China
http://dslab.lzu.edu.cn

Tools for GNU/Linux

- problem statement
- tools overview
- static instrumentation
- development life-cycle
- conclusion

tools are one of the strengths of GNU/Linux - evaluate the capabilities to see if it fits your project needs.

Problem Statement

- resource limitations
- no user monitoring
- applicaton scope insufficient
- failure analysis mandatory

Embedded systems are closer to clusters and servers than to desk-top systems - adjust your debugging !

Debugging Tools for Embedded GNU/Linux

Static Instrumentation
- KFI/KFT
- GCOV/Kernel GCOV
- LTT/LTTng
- GProf/Kernel GProf

Dynamic Instrument.
- Kprobes
- GDB Kernel Tracepoints
- KGDB Tracepoints
- Monitors

Breakpoint Debugging
- KGDB
- BDI 2000 / JTAG
- Lauterbach / JTAG

Dedicated Solutions
- User Mode Linux
- /proc interfaces
- Kernel Builtins
- Oprofile (hardware support)

Kernel Space Tools

- GDB/KGDB
- KFI/KFT
- GCOV/Kernel GCOV
- Oprofile
- LTT/LTTng
- Kprobes
- Kernel builtin debug extensions
- /proc interface

User Space Tools

- strace/ltrace/xtrace/mtrace
- Checkpoint Restart: i.e. BLCR
- LD_PRELOAD: i.e. libSegrault.so
- tons of malloc-debug-libs (i.e. njamd)
- BGCC/SSP gcc extensions
- *grind/DRD (?)
- standard unix tools

GDB/KGDB

- /proc/kcore to check the running kernel
- KGDB to debug the kernel
- UML under GDB control
- GDB over BDM (i.e. BDI 2000)
- GDB kernel tracepoints interface to kprobes
- KGDB tracepoints (very experimental at this point)

GDB will only help you if you have the experience of using it in the kernel - plan it in in your software development life cycle

Static Instrumentation

Properties of static insturmentation

- Pro
  - Relatively easy to use
  - Good system level relative timing (statistical)
  - Distortion of detailed timings
  - Supported in User Mode Linux
- Con
  - Relatively large overhead
  - Large data volume
  - Conflicting patches (unfortunately)
  - Interpretation requires experience with healthy systems

KFI/KFT

Kernel Function Instrumentation/Kernel Function Trace

- -finstrument-functions
- set config via /proc/kft
- get data via /proc/kft_data
- decode data via addr2sym/kd
- 50-200% overhead
- powerfull tool to understand the kernel

KFI/KFT will be helpfull in finding complex bugs - but only if you know how to use it before you need it !

GCOV / Kernel GCOV

- spaning tree of the kernel functions
- 64bit event counters per basic block
- interface via /proc/gcov
- Code coverage and branch prediction
- helps in system and kernel level performance assessment
- Allows optimization by recompiling with -fbranch-probabilities
- Hard to assign data to specific events/processes

GCOV is a standard profiling tool extended into kernel space with kernel GCOV - it is also available for UML

Oprofile

- built on performance counters (PMC)
- X86 and PPC
- precise for low-level HW-units
- low overhead
- inprecise with respect to timing and PID assignment

Oprofile is primarily used for performance tuning and locating of hardware artefacts (i.e. cach thrashing, BTB overload)

LTT/LTTng

- static instrumentation
- data via relayfs
- gives good system level overview
- good for detecting applicaton interaction problems
- low overhead if used **very** carfully - default setting not usable
- X86 centric but PPC port available (whith delay).

LTT is relatively easy to use - good starting point to locate "unknonw unknowns".

Kprobes

- breakpoint debugging in kernel mode
- Kprobes: insertion of arbitrary handlers
- Jprobes: handler at function entry
- return probes: handler at function return
- low overhead if used selectively
- relatively complex to use

Kernel Builtins

- scheduling statistics
- preemt timing measurement
- lock dependency checker
- vm debugging options

there is much much more here - and it is becomming more sophisticated all the time - Mainstream Linux has dramatically improved with the 2.6.X kernel series.

/proc Interface

- 0-overhead
- easy to use
- easy to integrate into user interface
- highly specialized information only
- secure tool for monitoring

SW-Lifecycle Tools Issues

- plan in tools in your regular test proceedures
- if you don't know the healthy system you can't read the pathological case !
- log your test and debug sessions !
- plan in the time for learning tools - you can't start using them when you need them

SW-Lifecycle Issues

- go to the target as late as possible
- keep your code arch independant by testing on two platforms
- automate the usage of Linux kernel debug tools
- integrate the tools into your product so you can get hig quality bug reports from your customers

Not every product will need this - but many more than currently are using the capabilities of GNU/Linux

Conclusion

- Learning GNU/Linux tools is an investment
- GNU/Linux tools allow better inspection than most commercial tools
- The GNU/Linux tools cover the entire spectrum from the applicatio layer into the kernel down to the hardware
- Dont wait to learn them until you need them !

Its free-software but you must invest in your engeneers so they can use it.