Agenda

• AutoFDO – the technique using trace to feedback into compilation for performance gains.

• Cross Triggering - Complex CoreSight trace configuration and use.

• Future directions of OpenCSD, Linux CoreSight Support and ARM Trace Hardware.
Feedback Directed Optimisation (FDO) - Overview

FDO uses profile data as an input to compiler optimisation to improve performance.

*AutoFDO* is an open source Google project generating profiles for FDO based on tracing the running system. CoreSight trace can now be used as the input for this analysis.

- **Gather**
  - Using production binaries

- **Analyse**
  - Using Open Source tools

- **Optimise**
  - Using standard compilers
Development of AutoFDO on ARM

• Initial contributions were made to the perf-opencsd repository to allow perf to analyse the collected trace and use in the perf --inject command – a key part of the process for AutoFDO. This was based on previously existing code that worked with Intel trace.

• Full ETM trace generates large amounts of data – which caused issues with storing, decoding and analysing the data. AutoFDO uses samples of execution to build profiles, so an ETM driver update was required.

• The ETM update used the counter and sequencer features of the ETM hardware to record only small slices of program execution – 1000’s of cycles captured per M cycles executed – known as strobing. This also conformed to the way the Intel trace was being used for AutoFDO.

• At present the strobing feature is not controlled by perf – but a record.sh script is provided to program this and start perf trace recording automatically.
Running an AutoFDO Session

Step

• Compile initial application – can be optimised but must contain sufficient debug info to match instructions to source lines.

• Capture trace data on the target system using the `record.sh` script. Command provided will collect 5000 cycles of trace every 50M cycles. Taskset ensures application stays on single core.

• Execution trace is converted to an instruction profile using perf built with OpenCSD trace decode library. The ‘inject’ command generates periodic instruction samples with branch histories.

Command Line

• When compiling use `--g1` or `--gmlt` for gcc, add `--fdebug-info-for-profiling` with llvm.

• `taskset -c 0 ./record.sh --strobe 5000 10000 28c06000.etr ./testapp arg1 arg2`

• `perf inject -i perf.data -o inj.data --itrace=i100000il`
Running an AutoFDO Session (2)

Step

• Instruction samples provided by perf inject are passed to the AutoFDO tools to generate source profiles for the compiler

• Recompile using the generated profiles.

Command Line

• *llvm*: `create_llvm_prof -binary=/path/to/testapp -profile=inj.data -out=testapp.llvmprof` or
  `gcc: create_gcov -binary=/path/to/testapp -profile=inj.data -gcov_version=1 -gcov=testapp.gcov`

• `gcc -O2 -fauto-profile=testapp.gcov -o testapp testapp.c` or
  `clang -O2 -fprofile-sample-use=testapp.llvmprof -o testapp testapp.c`
Results

- ARM ran tests to establish gains to be made using AutoFDO on ARM platforms.

- As AutoFDO is thought to provide more benefits to workloads with variable input data, SQLite was chosen. This has a built-in performance test (speedtest1) that was used to assess the effect of AutoFDO.

- The test platform was a HiKey 960 Board, running Android AOSP (March 2018). Compiler used was clang from NDK 17 beta (LLVM 6.0.1 r316199).

- Profile was generated from the –O2 build of the test application and used for AutoFDO.

<table>
<thead>
<tr>
<th>Test Size</th>
<th>Profile</th>
<th>Average Time</th>
<th>Vs Baseline (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>Baseline</td>
<td>148.77</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>A64</td>
<td>141.51</td>
<td>-4.88</td>
</tr>
<tr>
<td>500</td>
<td>X86</td>
<td>142.02</td>
<td>-4.54</td>
</tr>
<tr>
<td>500</td>
<td>PGO</td>
<td>143.71</td>
<td>-3.40</td>
</tr>
</tbody>
</table>
Resources needed to get started with AutoFDO

• AutoFDO tools: https://github.com/google/autofdo
  • Check for any pull requests relating to generating ARM profiles.

• Kernel with necessary patches:
  • 4.17 kernel has the perf updates required.
  • 4.19-rc1 kernel with latest CS drivers and ETM strobing patch on https://github.com/Linaro/perf-opencsd, afdo-etm-strobe branch
  • Backport trees for 4.9 and 4.14 kernels to be made available on ARM public site shortly.

• OpenCSD decoder on https://github.com/Linaro/OpenCSD;
  • This also contains detailed AutoFDO usage document – autofdo.md, and the record.sh script to program the ETM strobing.

• A compiler and target system!
Complex CoreSight Programming
Using CTI, ETM and STM
Embedded Cross Trigger (ECT) – CTI and CTM

CTI – Cross Trigger Interface; CTM – Cross Trigger Matrix

• Most CoreSight Components will have an associated CTI.
• A CTI may be connected to one or more components. Details of trigger connections are unusually implementation defined.
• v8 Arch mandates a CTI connected to the core and its associated ETM, and the trigger connections.
• The CTM connects all CTIs, which signal via 4 CTM channels.
CTI detailed operation

- Each input and output trigger line has its own register to enable connection to a channel (CTIINEN\(<n>\), CTIOUTEN\(<n>\))
- CTIGATE controls if active channels are transmitted to all other CTIs in the system
- Software can control channel state using CTIAPPSET, CTIAPPCLR and CTIAPPULSE
CoreSight Programming Scenarios (1)

Cross triggering to halt trace

- CTI can be used to halt collection in response to a software or hardware event.
- The ETR is programmed to flush and halt when the flush input from the associated CTI is set.
- The CTI associated with the PE and ETM can be programmed to send a signal via one of the channels to the ETR-CTI.
- The halt signal in the CTI can either be triggered by further programming the ETM to set the event output on an address, or the system software can write to the CTIAPPSET register to trigger the channel from an application / kernel device.
- Halting trace in this way ensures more of the trace at the point of the event is seen, and the latency associated with halting all the drivers is removed.
CoreSight Programming Scenarios (2)

STM hardware signal monitors

- STM has up to 256 hardware input signals that may be used to monitor signals in the system.
- Connections are implementation defined – so will differ between systems, if used at all.
- If connected, then STM can be programmed to insert packets into the trace stream if a monitored line changes state.
- Optionally can output a trigger to the CTI, that can then be used to halt trace on a hardware event.
CoreSight Programming Scenarios (3)

ETM Strobing

• The ETM has a number of hardware resources – comparators, counters, sequencer that may be programmed to control trace capture.
• perf will use address comparators if trace filters are used on the command line.
• The strobing effect – turning on the trace for short periods – is achieved by using a pair of counters, one to count the on “window” and one to count the number of “window” periods to be off.
• This has been achieved for AutoFDO by adding parameters to the driver to define the window and off periods, which are translated by the driver into the correct programming of the registers.
Programming practicalities

• Current advanced configurations need to be individually programmed using sysfs, then run capture using perf to collect the trace. Not well integrated.

• Large numbers of registers to be programmed mean adding these to a command line of perf not practical

• Current ETM strobe patch works, but uses general resources in a specific way. Adding new driver features each time we want to use the ETM resources in a new way is not scalable.

• Therefore we are planning a “complex configuration” feature for the CoreSight system. This will have the desired configuration registers in a file – so can be added to a perf command line easily, and will be usable standalone / for other applications.
Also available....

- **Post Mortem Trace and Debug.**
  Recovery of trace from ETB/ETF on crash. Also checks PC sample from debug registers.
  See training session from Connect@HKG18 by Leo Yan for more information (HKG18-TR14)

- **ftrace over STM.**
  CoreSight System Trace Macrocell (STM) driver integrates with Software Trace Module (STM!) in Linux kernel to allow ftrace output over trace. (as does Intel system trace hardware)
Future directions
Software Development

- **OpenCSD** – additional support for upcoming architecture increments – v8.3 and v8.4.

- **Perf** – Support for multicore tracing controlled by perf is in progress. (currently limited to `--per-thread` option only)

- **Linux Kernel Drivers:**
  - CTI drivers in progress – to be upstreamed.
  - Modifications required for multicore tracing.
  - Complex configuration support in CoreSight drivers

- **AutoFDO** – remaining patches will be superseded by complex configuration support in kernel. AutoFDO “HowTo” doc will be added to the OpenCSD repository.
CoreSight Base System Architecture (CoreSight-BSA)

• Hardware specification enabling designers and end users to exploit the full potential of hardware through standardised debug and profiling frameworks

• Focus on self-hosted access to Trace and CoreSight

• Linux perf “just works”

• AutoFDO “just works”
What is CoreSight-BSA?

• Defines the minimum hardware and firmware to ensure data gathering works
  • Defines functionality levels for Trace Generation, Trace Capture, STM and PE Debug.
    • ETM per-core
  • Capture in main memory, with options for dedicated memory
    • For main memory, the buffer is virtually-addressed, and needs translation
• Requires power to be managed by usual OS mechanisms
  • e.g. ACPI
• Requires CoreSight to be enabled and accessible
  • External debug does not need to be enabled
CoreSight-BSA level 0: current systems
CoreSight-BSA Level 2: Upcoming and Future systems.
To finish....
How do I enable trace on my platform?

- Does your platform have trace hardware?
- Is the trace hardware described in the device tree for the platform?

- Is the answer is “yes” to both, then you just need to build a kernel with the CoreSight drivers.
## Summary of CoreSight Trace features in Linux

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Trace Capture Using ETM + Sink and perf</td>
<td>Upstream.</td>
</tr>
<tr>
<td>Performance Enhancement using AutoFDO</td>
<td>Proof of concept available – partly upstreamed.</td>
</tr>
<tr>
<td>Cross Triggering and Complex configuration</td>
<td>Some patches available – in progress.</td>
</tr>
<tr>
<td>Software trace via STM</td>
<td>Connection to ftrace via linux STM, sysfs support for hardware STM connections.</td>
</tr>
<tr>
<td>Postmortem Debugging – Trace and Debug Registers</td>
<td>Upstream.</td>
</tr>
</tbody>
</table>
Resources

• OpenCSD trace decoder library: Debian distro or github: www.github.com/Linaro/opencsd

• perf-opencsd github repository – not yet upstream patches for new CoreSight features. www.github.com/Linaro/perf-opencsd

• CoreSight Mailing List: for questions about the OpenCSD decode library, and all things CoreSight, latest patches and fixes etc. coresight@lists.linaro.org

• For more on using perf on ARM and trace see the training session on Thursday: YVR18-416 – presented by Leo Yan and Daniel Thompson
Thank You
Danke
Merci
谢谢
ありがとうございます
Gracias
Kiitos
감사합니다
धन्यवाद
תודה