PMU Event Based Branches

Event Based Branches (EBBs) are a feature which allows the hardware to branch directly to a specified user space address when certain events occur.

The full specification is available in Power ISA v2.07:

https://www.power.org/documentation/power-isa-version-2-07/

One type of event for which EBBs can be configured is PMU exceptions. This document describes the API for configuring the Power PMU to generate EBBs, using the Linux perf_events API.

Terminology

Throughout this document we will refer to an "EBB event" or "EBB events". This just refers to a struct perf_event which has set the "EBB" flag in its attr.config. All events which can be configured on the hardware PMU are possible "EBB events".

Background

When a PMU EBB occurs it is delivered to the currently running process. As such EBBs can only sensibly be used by programs for self-monitoring.

It is a feature of the perf_events API that events can be created on other processes, subject to standard permission checks. This is also true of EBB events, however unless the target process enables EBBs (via mtspr(BESCR)) no EBBs will ever be delivered.

This makes it possible for a process to enable EBBs for itself, but not actually configure any events. At a later time another process can come along and attach an EBB event to the process, which will then cause EBBs to be delivered to the first process. It's not clear if this is actually useful.

When the PMU is configured for EBBs, all PMU interrupts are delivered to the user process. This means once an EBB event is scheduled on the PMU, no non-EBB events can be configured. This means that EBB events can not be run concurrently with regular 'perf' commands, or any other perf events.

It is however safe to run 'perf' commands on a process which is using EBBs. The kernel will in general schedule the EBB event, and perf will be notified that its events could not run.

The exclusion between EBB events and regular events is implemented using the existing "pinned" and "exclusive" attributes of perf_events. This means EBB events will be given priority over other events, unless they are also pinned. If an EBB event and a regular event are both pinned, then whichever is enabled first will be scheduled and the other will be put in error state. See the section below titled "Enabling an EBB event" for more information.

Creating an EBB event

To request that an event is counted using EBB, the event code should have bit 63 set.

EBB events must be created with a particular, and restrictive, set of attributes - this is so that they interoperate correctly with the rest of the perf_events subsystem.

An EBB event must be created with the "pinned" and "exclusive" attributes set. Note that if you are creating a group of EBB events, only the leader can have these attributes set.

An EBB event must NOT set any of the "inherit", "sample_period", "freq" or "enable_on_exec" attributes.

An EBB event must be attached to a task. This is specified to perf_event_open() by passing a pid value, typically 0 indicating the current task.

All events in a group must agree on whether they want EBB. That is all events must request EBB, or none may request EBB.

EBB events must specify the PMC they are to be counted on. This ensures userspace is able to reliably determine which PMC the event is scheduled on.

Enabling an EBB event

Once an EBB event has been successfully opened, it must be enabled with the perf_events API. This can be achieved either via the ioctl() interface, or the prctl() interface.

However, due to the design of the perf_events API, enabling an event does not guarantee that it has been scheduled on the PMU. To ensure that the EBB event has been scheduled on the PMU, you must perform a read() on the event. If the read() returns EOF, then the event has not been scheduled and EBBs are not enabled.

This behaviour occurs because the EBB event is pinned and exclusive. When the EBB event is enabled it will force all other non-

pinned events off the PMU. In this case the enable will be successful. However if there is already an event pinned on the PMU then the enable will not be successful.

Reading an EBB event

It is possible to read() from an EBB event. However the results are meaningless. Because interrupts are being delivered to the user process the kernel is not able to count the event, and so will return a junk value.

Closing an EBB event

When an EBB event is finished with, you can close it using close() as for any regular event. If this is the last EBB event the PMU will be deconfigured and no further PMU EBBs will be delivered.

EBB Handler

The EBB handler is just regular userspace code, however it must be written in the style of an interrupt handler. When the handler is entered all registers are live (possibly) and so must be saved somehow before the handler can invoke other code.

It's up to the program how to handle this. For C programs a relatively simple option is to create an interrupt frame on the stack and save registers there.

Fork

EBB events are not inherited across fork. If the child process wishes to use EBBs it should open a new event for itself. Similarly the EBB state in BESCR/EBBHR/EBBRR is cleared across fork().