big.LITTLE Cluster Migration Model

ARM Virtualizer Software Support for Debug and the Performance Management Unit

System Software on ARM®

Document number: ARM-EPM-018236

Copyright ARM Limited 2012

big.LITTLE Cluster Migration Model
ARM Virtualizer Software Support for Debug and the Performance Management Unit

Copyright © 2012 ARM Limited. All rights reserved.

Release information

The Change history table lists the changes made to this document.

Table 1 Change history

Date	Issue	Confidentiality	Change
18 April 2012	A-1	Confidential-Draft	First draft

Proprietary notice

Words and logos marked with ® and ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction 4

1.1 Scope 4
1.2 Purpose 4
1.3 Additional reading 4
1.3.1 ARM publications 4

2 Introduction 5

2.1 ARM Performance Monitoring Units 5
2.2 The big.LITTLE PMU Architecture 5

3 ARM Virtualizer Software support for the PMUs 11

3.1 ARM Virtualizer Software states 11
3.1.1 Transitions from one state to another 11
3.2 ARM Virtualizer Software interfaces 12
3.3 Usage restrictions 13
3.4 Access to specific PMU registers 13
3.4.1 Necessary implementation detail 14
3.4.2 Performance Monitor Control Register, PMCR 16
3.4.3 Event Counter Selection Register, PMSELR 17
3.4.4 Event Type Select Register, PMXEVTYPER 18
3.4.5 Count Enable Set Register, PMCNTENSET 19
3.4.6 Count Enable Clear Register, PMCNTENCLR 19
3.4.7 Cycle Count Register, PMCCNTR 20
3.4.8 Overflow Flag Status Register, PMOVSR 21
3.4.9 Event Count Register, PMXEVCNTR 21
3.4.10 Interrupt Enable Set Register, PMINTENSET 22
3.4.11 Interrupt Enable Clear Register, PMINTENCLR 23
3.5 Access to all PMU counters 24
3.5.1 Query size of memory to reserve (HVC_GET_COUNTERS_SIZE) 24
Usage: 24
3.5.2 Synchronize PMU counters (HVC_SYNC_PMU_COUNTERS) 24

4 Debug context migration 28

4.1 GDB example 28

5 Appendix A: Perf 30

6 Appendix B: Streamline 32

7 Glossary 34

Copyright © 2012 ARM. All rights reserved.
Confidential

1 Introduction

1.1 Scope

The scope of this document is limited to the Cluster Migration execution model implemented using the ARM Virtualizer Software v2.4 for a big.LITTLE system. The focus is on the Performance Monitor Unit and debugging. For the latter, the scope is limited to self hosted application debugging.

1.2 Purpose

The document describes the Performance Monitor Units on a big.LITTLE system and the interface to these units as provided by the ARM Virtualizer Software. In addition, the strategy for migrating debug context in order to support self-hosted application level debugging is described. This document enables the engineering of software tools that rely on the use of these units and the debug architecture on a big.LITTLE system that runs the ARM Virtualizer Software.

1.3 Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, `http://infocenter.arm.com`, for access to ARM documentation.

1.3.1 ARM publications

The following documents contain information relevant to this document:

1	Cortex™-A15 Technical Reference Manual	ARM DDI 0438D
2	Cortex-A7 MPCore Technical Reference Manual	ARM DDI 0464C
3	Performance Monitors v2 Supplement	ARM DDI 0457A
4	HVC Calling Conventions	DEN0020

Copyright © 2012 ARM. All rights reserved.
Confidential

2 Introduction

An ARM big.LITTLE system contains multiple Cortex-A15 and Cortex-A7 processors. These processors are instantiations of the ARMv7A architecture, but have very different micro-architectures: an energy efficient design in the case of the Cortex-A7 processor and a performance optimized design in the case of the Cortex-A15 processor. Micro-architectural differences between the two processor types are apparent in the system level architecture (CP15 space) including the processor-specific *Performance Monitoring Unit* (PMU).

There are two broad execution models for a big.LITTLE system:

1. Task Migration
2. MP

Task Migration has two sub-types: Cluster migration and CPU migration.

Cluster migration involves the use of a single processor cluster at any given moment, except very briefly during a cluster context switch to the other cluster. The aim is to stay resident on the energy efficient Cortex-A7 cluster while using the Cortex-A15 cluster opportunistically. This model is the focus of this document.

In the cluster migration execution model, the ARM Virtualizer Software is entrusted with masking micro-architectural differences between the Cortex-A15 and the Cortex-A7 processors. In addition, the software provides mechanisms to rapidly transfer state between the two processor clusters on demand. The ARM Virtualizer Software runs in privilege level PL2 (HYP Mode).

CPU migration involves the use of both clusters at the same time with restrictions on which processors are switched on at any given moment. CPUs are divided in CPU pairs (CPU0 on the Cortex-A15 and Cortex-A7processors, CPU1 on the Cortex-A15 and Cortex-A7 processors and so on). When using CPU migration only one CPU per processor pair can be used at the same time.

In the MP model, both clusters may be used simultaneously under OS scheduler control.

These different execution models each have their own advantages and disadvantages.

This document specifies the usage of the ARM PMUs on a big.LITTLE system used with the Task Migration execution model using the ARM Virtualizer Software.

The document also specifies the debug context that is migrated between the two clusters.

2.1 ARM Performance Monitoring Units

The Cortex-A15 and Cortex-A7 processors include instantiations of the ARM PMUv2 architecture. The PMU instance gathers various run-time statistics on the operation of the processor and the memory system which is useful for understanding the performance of the processor and for debugging or profiling code.

ARM recommends that readers of this document first familiarize themselves with the PMU architecture in general by referring to the ARM Architecture Reference Manual.

2.2 The big.LITTLE PMU Architecture

This document considers a symmetric (in terms of the number of CPUs on each side) big.LITTLE system with a Cortex-A15 cluster and a Cortex-A7 cluster.

The PMU implementation on a Cortex-A15 processor differs from that of a Cortex-A7 processor PMU.

Specifically:

Copyright © 2012 ARM. All rights reserved.
Confidential

a. The Cortex-A15 processor has six PMU counters while the Cortex-A7 processor has four PMU counters.

b. The Cortex-A15 and Cortex-A7 processors can count different events. For a complete list of which events are supported on each processor refer to Table 2.

These differences pose challenges for all the envisioned execution models. For example, a requirement might be that events be counted only on a given processor type, or that events be counted separately for each processor cluster across a switch. Alternatively, migrating events across cluster switches might be required, with event counts being summed across runs on multiple clusters.

In the case of Task migration, the Virtualizer Software aims to provide a best effort solution to cater to these problems while still remaining flexible.

The ARM Virtualizer Software ensures that processors on all clusters have the same number of visible counters. It implements different states to support different profiler tool behaviors. It also exports two different interfaces, which allow access to PMU registers, events and counters and also changing the ARM Virtualizer software state as appropriate.

Table 2 lists all supported PMUv2 architecture events in the system. Events in black are common among all the clusters, while events in red are different at least on a cluster.

Event numbers which are not listed in the following table must be considered as unimplemented and using them will result in undefined behavior.

Events marked with a – (dash) should be considered as unimplemented and using them will result in undefined behavior.

Specifically for the Cortex-A15 and Cortex-A7 processors:

1. Every event number in the range 0x00 to 0x3F is defined by the ARM Architecture Reference Manual (**Type 1**).
2. Every event number in the range 0x40 to 0x8F is defined by the ARM recommendations for the use of the IMPLEMENTATION DEFINED event numbers, as defined by issue C of the ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. (**Type 2**).
3. Every event number in the range 0x90 to 0xFF counts the event defined in the processor implementation’s technical reference manual (TRM), or an undocumented event (**Type 3**).

These event types and their treatment by the ARM Virtualizer Software are explained in later sections.

The Cortex-A15 and the Cortex-A7 processors implement the following PMUv2 architecture events.

Table 2: PMU detail

Event number	Cortex-A15 Event description (as defined in Cortex-A15 TRM)	Cortex-A7 Event description (as defined in Cortex-A7 TRM)	Type of event
0x00	Instruction architecturally executed (condition check pass) - Software increment	Software increment. The register is incremented only on writes to the Software Increment Register.	Type 1
0x01	Level 1 instruction cache refill	Instruction fetch that causes a refill at (at least) the lowest level of instruction or unified cache. Includes the speculative linefills in the count.	Type 1
0x02	Level 1 instruction TLB refill	Instruction fetch that causes a TLB refill at (at least) the lowest level of TLB. Includes the speculative requests in the count.	Type 1
0x03	Level 1 data cache refill	Data read or write operation that causes a refill at (at least) the lowest level of data or unified cache. Counts the number of allocations performed in the Data Cache because of a read or a write.	Type 1

Copyright © 2012 ARM. All rights reserved.
Confidential

0x04	Level 1 data cache access	Data read or write operation that causes a cache access at (at least) the lowest level of data or unified cache. This includes speculative reads.	Type 1
0x05	Level 1 data TLB refill	Data read or write operation that causes a TLB refill at (at least) the lowest level of TLB. This does not include micro TLB misses because of PLD, PLI, CP15 Cache operation by MVA and CP15 VA to PA operations.	Type 1
0x06	-	Data read architecturally executed. Counts the number of data read instructions accepted by the Load Store Unit. This includes counting the speculative and aborted LDR/LDM, and the reads because of the SWP instructions.	Type 1
0x07	-	Data write architecturally executed. Counts the number of data write instructions accepted by the Load Store Unit. This includes counting the speculative and aborted STR/STM, and the writes because of the SWP instructions.	Type 1
0x08	Instruction architecturally executed	Instruction architecturally executed.	Type 1
0x09	Exception taken	Exception taken. Counts the number of exceptions architecturally taken.	Type 1
0x0A	Instruction architecturally executed (condition check pass) - Exception return	Exception return architecturally executed.	Type 1
0x0B	Instruction architecturally executed (condition check pass) - Write to CONTEXTIDR	Change to ContextID retired. Counts the number of instructions architecturally executed writing into the ContextID Register.	Type 1
0x0C	-	Software change of PC.	Type 1
0x0D	-	Immediate branch architecturally executed (taken or not taken). This includes the branches which are flushed due to a previous load/store which aborts late.	Type 1
0x0E	-	Procedure return (other than exception returns) architecturally executed.	Type 1
0x0F	-	Unaligned load-store.	Type 1
0x10	Mispredicted or not predicted branch speculatively executed	Branch mispredicted/not predicted. Counts the number of mispredicted or not-predicted branches executed. This includes the branches which are flushed because of a previous load/store which aborts late.	Type 1
0x11	Cycle	Cycle counter.	Type 1
0x12	Predictable branch speculatively executed	Branches or other change in program flow that could have been predicted by the branch prediction resources of the processor. This includes the branches which are flushed because of a previous load/store which aborts late.	Type 1
0x13	Data memory access	Data memory access.	Type 1
0x14	Level 1 instruction cache access	Instruction Cache access.	Type 1
0x15	Level 1 data cache Write-Back	Data cache eviction.	Type 1
0x16	Level 2 data cache access	Level 2 data cache access	Type 1
0x17	Level 2 data cache refill	Level 2 data cache refill	Type 1

Copyright © 2012 ARM. All rights reserved.
Confidential

0x18	Level 2 data cache Write-Back	Level 2 data cache write-back. Data transfers made as a result of a coherency request from the Level 2 caches to outside of the Level 1 and Level 2 caches are not counted. Write-backs made as a result of CP15 cache maintenance operations are counted.	Type 1
0x19	Bus access	Bus accesses. Single transfer bus accesses on either of the ACE read or write channels might increment twice in one cycle if both the read and write channels are active simultaneously.Operations that utilise the bus that do not explicitly transfer data, such as barrier or coherency operations are counted as bus accesses.	Type 1
0x1A	Local memory error	-	Type 1
0x1B	Instruction speculatively executed	-	Type 1
0x1C	Instruction architecturally executed (condition check pass) - Write to translation table base	-	Type 1
0x1D	Bus cycle	Bus cycle	Type 1
0x40	Level 1 data cache access – Read	-	Type 2
0x41	Level 1 data cache access – Write	-	Type 2
0x42	Level 1 data cache refill – Read	-	Type 2
0x43	Level 1 data cache refill – Write	-	Type 2
0x46	Level 1 data cache Write-Back - Victim	-	Type 2
0x47	Level 1 data cache Write-Back - Cleaning and coherency	-	Type 2
0x48	Level 1 data cache invalidate	-	Type 2
0x4C	Level 1 data TLB refill – Read	-	Type 2
0x4D	Level 1 data TLB refill – Write	-	Type 2
0x50	Level 2 data cache access - Read	-	Type 2
0x51	Level 2 data cache access - Write	-	Type 2
0x52	Level 2 data cache refill - Read	-	Type 2
0x53	Level 2 data cache refill - Write	-	Type 2
0x56	Level 2 data cache Write-Back - Victim	-	Type 2
0x57	Level 2 data cache Write-Back - Cleaning and coherency	-	Type 2
0x58	Level 2 data cache invalidate	-	Type 2
0x60	Bus access - Read	Bus access, read	Type 2
0x61	Bus access - Write	Bus access, write	Type 2
0x62	Bus access - Normal	-	Type 2
0x63	Bus access - Not normal	-	Type 2

Copyright © 2012 ARM. All rights reserved.
Confidential

0x64	Bus access - Normal	-	Type 2
0x65	Bus access - Peripheral	-	Type 2
0x66	Data memory access - Read	-	Type 2
0x67	Data memory access - Write	-	Type 2
0x68	Unaligned access - Read	-	Type 2
0x69	Unaligned access - Write	-	Type 2
0x6A	Unaligned access	-	Type 2
0x6C	Exclusive instruction speculatively executed - LDREX	-	Type 2
0x6D	Exclusive instruction speculatively executed - STREX pass	-	Type 2
0x6E	Exclusive instruction speculatively executed - STREX fail	-	Type 2
0x70	Instruction speculatively executed - Load	-	Type 2
0x71	Instruction speculatively executed - Store	-	Type 2
0x72	Instruction speculatively executed - Load or store	-	Type 2
0x73	Instruction speculatively executed - Integer data processing	-	Type 2
0x74	Instruction speculatively executed - Advanced SIMD	-	Type 2
0x75	Instruction speculatively executed - VFP	-	Type 2
0x76	Instruction speculatively executed - Software change of the PC	-	Type 2
0x78	Branch speculatively executed - Immediate branch	-	Type 2
0x79	Branch speculatively executed - Procedure return	-	Type 2
0x7A	Branch speculatively executed - Indirect branch	-	Type 2
0x7C	Barrier speculatively executed - ISB	-	Type 2
0x7D	Barrier speculatively executed - DSB	-	Type 2
0x7E	Barrier speculatively executed – DMB	-	Type 2
0x86	-	IRQ exception taken.	Type 2
0x87	-	FIQ exception taken	Type 2
0xC0	-	External memory request	Type 3
0xC1	-	Non-cacheable external memory request	Type 3

Copyright © 2012 ARM. All rights reserved.
Confidential

0xC2	-	Linefill prefetch.	Type 3
0xC3	-	Prefetch linefill dropped	Type 3
0xC4	-	Entering read allocate mode.	Type 3
0xC5	-	Read allocate mode.	Type 3
0xC7	-	ETM Ext Out[0].	Type 3
0xC8	-	ETM Ext Out[1].	Type 3
0xC9	-	Data Write operation that stalls the pipeline because the store buffer is full.	Type 3
0xCA	-	Data snooped from other processor. This event counts memory-read operations that read data from another processor within the local Cortex-A7 cluster, rather than accessing the L2 cache or issuing an external read. It increments on each transaction, rather than on each beat of data.	Type 3

 Copyright © 2012 ARM. All rights reserved.
Confidential

3 ARM Virtualizer Software support for the PMUs

As defined in paragraph 2.2, the ARM Virtualizer Software enables the use of the PMUs in different ways as per need. To support this it exposes two different interfaces (as described in paragraph 3.2), and implements three different states of operation (as described in paragraph 3.1).

Using these mechanisms, the PMUs may be used conventionally, via MCR/MRC instructions or by specialized APIs exposed by the ARM Virtualizer Software.

3.1 ARM Virtualizer Software states

The ARM Virtualizer Software supports three different states:

State 1: Saving and restoring PMU context across a cluster switch is disabled.

State 2: There is only one PMU context. On a cluster switch, it is saved on the outbound cluster and restored on the inbound cluster.

State 3: There are two PMU contexts, one for each cluster. The outbound cluster PMU context is saved on the outbound cluster before the switch and the inbound cluster PMU context is restored on the inbound cluster after the switch.

The system boots in State 1.

The use of these states enables different strategies for using the PMUs.

When in State 1 the ARM Virtualizer Software will trap all MRC/MCR instructions which attempt to access any PMU register and will not save and restore any PMU register across a cluster switch. PMUs are not used in this state but the use of the PMUs via MRC/MCR instructions is the trigger to switch to state 2.

When in state 2 the ARM Virtualizer Software disables traps upon access to any PMU register through MRC/MCR instructions. It enables save/restore and migration of all PMU registers across a cluster switch. When in this state, the ARM Virtualizer Software maintains only one PMU context that is switched between clusters.

In State 2, if a cluster switch occurs, the PMU context of the outbound cluster is saved before the switch and the same PMU context is restored on the inbound cluster after the cluster switch without any specific checks. This permits ‘raw’ migration of events between processor clusters. The onus is on the user to ensure that the events in question are suitable.

When in State 3 the ARM Virtualizer Software enables traps to all PMU registers through MRC/MCR instructions. It enables save/restore of PMU contexts across a cluster switch, without migrating values. When in this state the ARM Virtualizer Software keeps one separate PMU context for each cluster.

In State 3, if a cluster switch occurs, the PMU context of the outbound cluster is saved on the outbound cluster before the switch. The PMU context of the inbound cluster is restored on the inbound cluster after the switch.

3.1.1 Transitions from one state to another

Transition from State 1 to State 2 is triggered by executing any MCR/MRC instruction accessing any PMU register. The instruction execution will be trapped by the ARM Virtualizer Software, which will then switch the system to State 2.

Transition from State 1 to State 3 is triggered by executing the HVC_PMU_SWITCH (Table 3) passing 1 as parameter. The ARM Virtualizer Software will then switch the system to State 3.

Copyright © 2012 ARM. All rights reserved.
Confidential

Transition from State 2 to State 3 is triggered by executing the HVC_PMU_SWITCH (Table 3) passing 1 as parameter. The ARM Virtualizer Software will then switch the system to State 3.

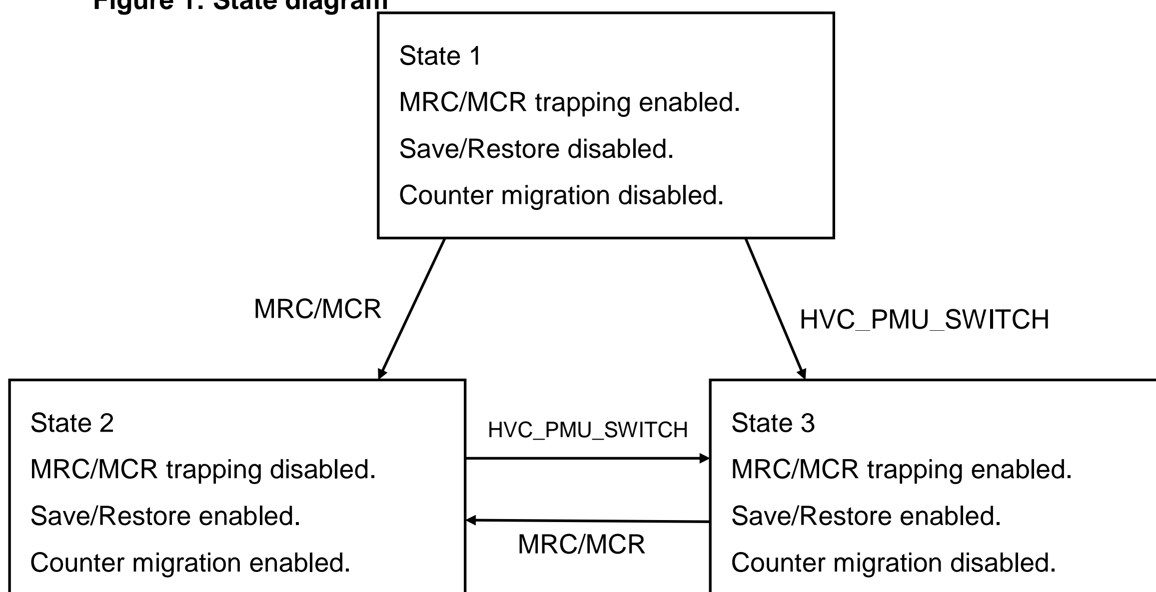
Transition from State 3 to State 1 is triggered by executing HVC_PMU_SWITCH (Table 3) passing 0 as parameter. The ARM Virtualizer Software will then switch the system to State 1.

Transition from State 3 to State 2 is triggered by executing any MCR/MRC instruction accessing any PMU register.

Transition from State 2 to State 1 is triggered by executing HVC_PMU_SWITCH (Table 3) passing 0 as parameter. The system will then switch to State 1.

The following diagram shows the states as implemented by the ARM Virtualizer Software:

Figure 1: State diagram



3.2 ARM Virtualizer Software interfaces

The ARM Virtualizer Software provides two mutually exclusive ways to access the PMUs on a big.LITTLE system from PL1 modes:

1. Access to specific PMU registers via MCR/MRC instructions (**Type A).**
2. Access to specific PMU registers and to all PMU events and counters via a specified API **(Type B).**

There are restrictions on the simultaneous use of these mechanisms. See section 3.3 for details on this and how the migration and availability of events are handled.

For point 2 above, the ARM Virtualizer Software provides APIs that follow the condensed HVC calling convention as specified in the HVC calling conventions specification [4]. Details regarding the hypercall instruction and specific registers used in various HVC calls described here are provided by that specification.

Copyright © 2012 ARM. All rights reserved.
Confidential

3.3 Usage restrictions

The Type A interface enables profiler tools to program any counter on the current cluster with some restrictions in place.

1. All event types (Type 1, Type 2 and Type 3 – see paragraph 2.2) can be programmed on counters on processors in any cluster but the event will only be counted if it is implemented by the processor in question. If the event is not implemented, the event will not be counted, even though the programming action succeeds.
2. For Type 1 and Type 2, if the event is implemented on processors on both clusters, then it is guaranteed that the counters count the same event for the event number in question.
3. The above points imply that for Type 1 and Type 2 events that are supported by processors on both clusters, switching clusters will result in meaningful migration of the event counts.
4. A Type 3 event, by definition, is only implemented by one of the two processor types and switching clusters will not result in a meaningful migration of the event counts, even if the event number happens to be the same.

There are no such restrictions when using the Type B (paragraph 3.2) interface.

3.4 Access to specific PMU registers

The ARM Virtualizer Software doesn't trap any access to PMU registers via MRC/MCR instructions (any traps are only for triggering state changes). This is the conventional interface to the PMUs and is known as the Type A interface.

The ARM Virtualizer Software also exposes another set of interfaces that are cumulatively referred to as the Type B interface. The Type B interface has two API sets, one for access to specific PMU registers on specific clusters and one for access to all PMU registers on all clusters.

The illustrative C prototype of the API call for a read operation via the 'specific' Type B interface API is :

unsigned long hvc (HVC_<REG>_READ, int cluster_id);

where:

- **HVC_<REG>_READ** specifies the register In question (see Table 4)
- **cluster_id** is integer 0 (zero) for the A15 cluster and integer 1 (one) for the A7 cluster

  All addresses passed to the ARM Virtualizer software must be physical addresses.

- The return value is stored in register r0
- The HVC always returns the value of the register on the CPU which is performing the hypervisor call. The cpu_id is then not needed and implicitly set to the CPU executing the HVC.

There's only one exception, when reading the cycle counter register:

u64 hvc(HVC_CCNT_READ, int cluster_id);

The cycle counter value returned is always a 64 bits value. The least significant word is returned in r0 and the most significant word in r1.

The illustrative C prototype of the API call for a write operation via the 'specific' Type B interface API is:

Copyright © 2012 ARM. All rights reserved.
Confidential

void hvc (HVC_<REG>_WRITE, int cluster_id, unsigned long value);

where:

- **HVC_<REG>_WRITE** specifies the register in question (see Table 4)
- **cluster_id** is integer 0 (zero) for the Cortex-A15 processor cluster and integer 1 (one) for the Cortex-A7 processor cluster
- **value** is a unsigned long denoting the value to be written.

There's only one exception, when writing the cycle counter register:

void hvc(HVC_CCNT_WRITE, int_cluster_id, u64 value);

The least significant word must be written in r0 and the most significant word in r1.

3.4.1 Necessary implementation detail

The three supported mechanisms provide varying degrees of flexibility and detail for interaction with the PMUs. However, there are restrictions on the simultaneous use of these mechanisms. In order to understand these restrictions, some implementation background is necessary.

The ARM Virtualizer Software has the concept of a boot cluster and a secondary cluster. The boot cluster is also known as the target cluster and the secondary cluster is also known as the host cluster. A host cluster emulates the target. For example, if the ARM Virtualizer software is built to ensure that the boot cluster (target) is the Cortex-A15 cluster, the intention is to run software built for the Cortex-A15 processor on this cluster. When the software stack is switched to the Cortex-A7 cluster (the secondary or host cluster), the ARM Virtualizer software will emulate a Cortex-A15 cluster thus allowing Cortex-A15 processor-specific software accesses to be trapped and handled accordingly.

Type A detail

The Type A PMU API is intended to support existing PMU tools which access PMU registers through MRC/MCR instructions such as the Linux Perf tool without the need to modify/extend the tool. This enables the use of Perf on a cluster switching system today. In the future, the perf tool shall be enhanced to support multiple PMU types (for the Cortex-A15 and Cortex-A7 processors) on a big.LITTLE system.

The user has full access to all PMU events and counters on both clusters. Particular care must be taken when choosing events to measure in order to get sensible data (following rules described in paragraph 3.3)

Type B detail

The Type B interface provides an HVC API for access to specific PMU registers across the system (or all PMU counters across the whole system), irrespective of the type of the boot cluster or the secondary cluster. The intention here is not to propagate events and counts across a switch but to program the different PMU types independently. To understand this interface, consider the following:

1. To support this interface, the ARM Virtualizer software does not need to switch PMU state across a cluster switch.
2. When this API is used to program counters on the currently inactive cluster, the programming action occurs only when a switch to that cluster takes place.
3. If this API is used to access an inactive cluster, the values returned are either values captured when switching away from that cluster in the past, or zero (in the case when the cluster was never enabled).

Copyright © 2012 ARM. All rights reserved.
Confidential

To enable the Type B interface, the following HVC is provided:

Table 3

HVC_PMU_SWITCH	0x9000_1100

The HVC_PMU_SWITCH moves the system to state 3 if the parameter in r1 is 1, or it moves the system to state 1 if parameter in r1 is 0.

Simultaneous use restrictions on Type A and Type B interfaces

The two different interfaces types require different implementation pathways in the ARM Virtualizer software, given that the MRC/MCR interface (Type A) requires migration of saved registers across a cluster switch and HVC interface (Type B) does not.

Using a specific interface triggers a change in the state of the ARM Virtualizer software. Interleaving instructions which access the PMUs both through MRC/MCR instructions and HVC instructions will cause undefined behavior and should be avoided

This means that if, say, Streamline uses Type B and Perf uses Type A, they must not be used simultaneously and this responsibility falls on the user of the profiler tools.

Copyright © 2012 ARM. All rights reserved.
Confidential

Table 4: PMU register encoding

HVC_PMCR_READ	0x9000_1000
HVC_PMCR_WRITE	0x9000_1001
HVC_PMSELR_READ	0x9000_1002
HVC_PMSELR_WRITE	0x9000_1003
HVC_PMXEVTYPER_READ	0x9000_1004
HVC_PMXEVTYPER_WRITE	0x9000_1005
HVC_PMCNTENSET_READ	0x9000_1006
HVC_PMCNTENSET_WRITE	0x9000_1007
HVC_PMCNTENCLR_READ	0x9000_1008
HVC_PMCNTENCLR_WRITE	0x9000_1009
HVC_PMCCNTR_READ	0x9000_100A
HVC_PMCCNTR_WRITE	0x9000_100B
HVC_PMOVSR_READ	0x9000_100C
HVC_PMOVSR_WRITE	0x9000_100D
HVC_PMXEVCNTR_READ	0x9000_100E
HVC_PMXEVCNTR_WRITE	0x9000_100F
HVC_PMINTENSET_READ	0x9000_1010
HVC_PMINTENSET_WRITE	0x9000_1011
HVC_PMINTENCLR_READ	0x9000_1012
HVC_PMINTENCLR_WRITE	0x9000_1013

The following subsections describe the specific accesses.

3.4.2 Performance Monitor Control Register, PMCR

The PMCR provides details of the performance monitor implementation, including the number of counters implemented, and configures and controls the counters.

Table 5: PMCR

MRC p15,0,<Rt>,c9,c12,0	Read PMCR on the running cluster

Copyright © 2012 ARM. All rights reserved.
Confidential

hvc (HVC_PMCR_READ, 0);	Read PMCR on the Cortex-A15 processor cluster
hvc (HVC_PMCR_READ, 1);	Read PMCR on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9,c12,0	Write PMCR on the running cluster
hvc (HVC_PMCR_WRITE, 0, pmcr);	Write PMCR on the Cortex-A15 processor cluster
hvc (HVC_PMCR_WRITE, 1, pmcr);	Write PMCR on the Cortex-A7 processor cluster

Example:

To read the PMCR register:

```
inline unsigned long armv7_pmnc_read(int cluster_id)
{
        unsigned long val;
        val = hvc(HVC_PMCR_READ, cluster_id);
        return val;
}
```

To write the PMCR register:

```
inline void armv7_pmnc_write(int cluster_id, unsigned long val)
{
        hvc(HVC_PMCR_WRITE, cluster_id, val);
}
```

3.4.3 Event Counter Selection Register, PMSELR

The PMSELR selects the current event counter.

Table 6: PMSELR

MRC p15,0,<Rt>,c9,c12,5	Read PMSELR on the running cluster
hvc (HVC_PMSELR_READ, 0);	Read PMSELR on the Cortex-A15 processor cluster
hvc (HVC_PMSELR_READ, 1);	Read PMSELR on the Cortex-A7 processor cluster

Copyright © 2012 ARM. All rights reserved.
Confidential

MCR p15,0,<Rt>,c9,c12,5	Write PMSELR on the running cluster
hvc (HVC_PMSELR_WRITE, 0, pmselr);	Write PMSELR on the Cortex-A15 processor cluster
hvc (HVC_PMSELR_WRITE, 1, pmselr);	Write PMSELR on the Cortex-A7 processor cluster

Example:

```
static inline void armv7_pmnc_select_counter(int cluster_id,
unsigned long regval)
 {
        hvc(HVC_PMSELR_WRITE, cluster_id, regval);
 }
```

3.4.4 Event Type Select Register, PMXEVTYPER

The PMXEVTYPER selects the events that an event counter is to count.

Table 7: PMXEVTYPER

MRC p15,0,<Rt>,c9,c13,1	Read PMXEVTYPER on the running cluster
hvc (HVC_PMXEVTYPER_READ, 0);	Read PMXEVTYPER on the Cortex-A15 processor cluster
hvc (HVC_PMXEVTYPER _READ, 1);	Read PMXEVTYPER on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9,c13,1	Write PMXEVTYPER on the running cluster
hvc (HVC_PMXEVTYPER _WRITE, 0, pmxevtyper);	Write PMXEVTYPER on the Cortex-A15 processor cluster
hvc (HVC_PMXEVTYPER _WRITE, 1, pmxevtyper);	Write PMXEVTYPER on the Cortex-A7 processor cluster

Example:

In a system with N counters, cnt can assume values from 0 to N - 1

To select the event type:

```
static inline void armv7_pmnc_write_evtsel(int cluster_id, unsigned
int cnt, unsigned long regval)
{
        hvc(HVC_PMXEVTYPER_WRITE, cluster_id, regval);
```


Copyright © 2012 ARM. All rights reserved.
Confidential

```
}
```

3.4.5 Count Enable Set Register, PMCNTENSET

The PMCNTENSET enables the Cycle Count Register (PMCCNTR) and any implemented event counters. Reading this register indicates which counters are enabled.

Table 8: PMCNTENSET

MRC p15,0,<Rt>,c9,c12,1	Read PMCNTENSET on the running cluster
hvc (HVC_ PMCNTENSET_READ, 0);	Read PMCNTENSET on the Cortex-A15 processor cluster
hvc (HVC_ PMCNTENSET_READ, 1);	Read PMCNTENSET on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9,c12,1	Write PMCNTENSET on the running processor cluster
hvc (HVC_ PMCNTENSET_WRITE, 0, pmcntenset);	Write PMCNTENSET on the Cortex-A15 processor cluster
hvc (HVC_ PMCNTENSET_WRITE, 1, pmcntenset);	Write PMCNTENSET on the Cortex-A7 processor cluster

Example:

```
static inline void armv7_pmnc_enable_counter(unsigned long regval)
{
    hvc(HVC_PMCNTENSET_WRITE, cluster_id, regval);
}
```

3.4.6 Count Enable Clear Register, PMCNTENCLR

The PMCNTENCLR disables the Cycle Count Register, PMCCNTR, and any implemented event counters. Reading this register shows which counters are enabled (like PMCNTENSET).

Table 9: PMCNTENCLR

MRC p15,0,<Rt>,c9,c12,2	Read PMCNTENCLR on the running cluster
hvc (HVC_PMCNTENCLR_READ, 0);	Read PMCNTENCLR on the Cortex-A15 processor cluster
hvc (HVC_PMCNTENCLR_READ, 1);	Read PMCNTENCLR on the Cortex-A7

Copyright © 2012 ARM. All rights reserved.
Confidential

	processor cluster

MCR p15,0,<Rt>,c9,c12,2	Write PMCNTENCLR on the running cluster
hvc (HVC_ PMCNTENCLR_WRITE, 0, pmcntenclr);	Write PMCNTENCLR on the Cortex-A15 processor cluster
hvc (HVC_ PMCNTENCLR_WRITE, 1, pmcntenclr);	Write PMCNTENCLR on the Cortex-A7 processor cluster

Example:

```
static inline void armv7_pmnc_disable_counter(unsigned long regval)
{
    hvc(HVC_PMCNTENCLR_WRITE, cluster_id, regval);
}
```

3.4.7 Cycle Count Register, PMCCNTR

The PMCCNTR counts processor clock cycles.

Table 10: PMCCNTR

MRC p15,0,<Rt>,c9,c13,0	Read PMCCNTR on the running cluster
hvc (HVC_PMCCNTR_READ, 0);	Read PMCCNTR on the Cortex-A15 processor cluster
hvc (HVC_PMCCNTR_READ, 1);	Read PMCCNTR on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9,c13,0	Write PMCCNTR on the running cluster
hvc (HVC_PMCCNTR_WRITE, 0, pmcntenclr);	Write PMCCNTR on the Cortex-A15 processor cluster
hvc (HVC_PMCCNTR_WRITE, 1, pmcntenclr);	Write PMCCNTR on the Cortex-A7 processor cluster

Example:

```
inline u64 armv7_ccnt_read(int cluster_id)
{
        u64 val;

      val = hvc(HVC_PMCCNTR_READ, cluster_id);
```


Copyright © 2012 ARM. All rights reserved.
Confidential

```
        return val;
}
```

3.4.8 Overflow Flag Status Register, PMOVSR

The PMOVSR holds the state of the overflow bits for the Cycle Count Register, PMCCNTR, and each of the implemented event counters. Software must write to this register to clear these bits.

Table 11: PMOVSR

MRC p15,0,<Rt>,c9, c12,3	Read PMOVSR on the running cluster
hvc (HVC_PMOVSR_READ, 0);	Read PMOVSR on the Cortex-A15 processor cluster
hvc (HVC_PMOVSR_READ, 1);	Read PMOVSR on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9, c12,3	Write PMOVSR on the running cluster
hvc (HVC_PMOVSR_WRITE, 0);	Write PMOVSR on the Cortex-A15 processor cluster
hvc (HVC_PMOVSR_WRITE, 1);	Write PMOVSR on the Cortex-A7 processor cluster

Example:

To reset the interrupt:

```
inline unsigned long armv7_pmnc_reset_interrupt(void)
{
        // Get and reset overflow status flags
        unsigned long flags;
        hvc(HVC_PMOVSR_READ, cluster_id, &flags);
        flags &= 0x8000003f;
        hvc(HVC_PMOVSR_WRITE, cluster_id, flags);
        return flags;
}
```

3.4.9 Event Count Register, PMXEVCNTR

The PMXEVCNTR is used to read or write the value of the current event counter, PMNx.

Table 12: PMXEVCNTR

MRC p15, 0, <Rt>, c9, c13, 2	Read PMXEVCNTR on the running cluster

Copyright © 2012 ARM. All rights reserved.
Confidential

hvc (HVC_PMXEVCNTR_READ, 0);	Read PMXEVCNTR on the Cortex-A15 processor cluster
hvc (HVC_PMXEVCNTR_READ, 1);	Read PMXEVCNTR on the Cortex-A7 processor cluster

MCR p15, 0, <Rt>, c9, c13, 2	Write PMXEVCNTR on the running cluster
hvc (HVC_PMXEVCNTR_WRITE, 0, pmxevcntr);	Write PMXEVCNTR on the Cortex-A15 processor cluster
hvc (HVC_PMXEVCNTR_WRITE, 1, pmxevcntr);	Write PMXEVCNTR on the Cortex-A7 processor cluster

```
inline unsigned long armv7_cntn_read(int cluster_id)
{
        unsigned long val;

        val = hvc(HVC_ PMXEVCNTR _READ, cluster_id);

        return val;
 }
```

3.4.10 Interrupt Enable Set Register, PMINTENSET

PMINTENSET enables the generation of interrupt requests on overflows from:

- the Cycle Count Register, PMCCNTR
- each implemented event counter, PMNx.

Reading the register shows which overflow interrupt requests are enabled.

Table 13: PMINTENSET

MRC p15,0,<Rt>,c9,c14,1	Read PMINTENSET on the running cluster
hvc (HVC_PMINTENSET_READ, 0);	Read PMINTENSET on the Cortex-A15 processor cluster
hvc (HVC_PMINTENSET_READ, 1);	Read PMINTENSET on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9,c14,1	Write PMINTENSET on the running cluster
hvc (HVC_PMINTENSET_WRITE, 0, pmintenset);	Write PMINTENSET on the Cortex-A15 processor cluster

Copyright © 2012 ARM. All rights reserved.
Confidential

hvc (HVC_PMINTENSET_WRITE, 1, pmintenset);	Write PMINTENSET on the Cortex-A7 processor cluster

Example

```
static inline void armv7_pmnc_enable_interrupt(int cluster_id, unsigned long regval)
{
        hvc(HVC_PMINTENSET_WRITE, cluster_id, regval);
}
```

3.4.11 Interrupt Enable Clear Register, PMINTENCLR

The PMINTENCLR disables the generation of interrupt requests on overflows from:

- the Cycle Count Register, PMCCNTR
- each implemented event counter, PMNx.

Reading the register shows which overflow interrupt requests are enabled.

Table 14: PMINTENCLR

MRC p15,0,<Rt>,c9,c14,2	Read PMINTENCLR on the running cluster
hvc (HVC_PMINTENCLR_READ, 0);	Read PMINTENCLR on the Cortex-A15 processor cluster
hvc (HVC_PMINTENCLR_READ, 1);	Read PMINTENCLR on the Cortex-A7 processor cluster

MCR p15,0,<Rt>,c9,c14,2	Write PMINTENCLR on the running cluster
hvc (HVC_PMINTENCLR_WRITE, 0, pmintenclr);	Write PMINTENCLR on the Cortex-A15 processor cluster
hvc (HVC_PMINTENCLR_WRITE, 1, pmintenclr);	Write PMINTENCLR on the Cortex-A7 processor cluster

Example

To disable overflow interrupt:

```
static inline void armv7_pmnc_disable_interrupt(int cluster_id, unsigned long regval)
{
                hvc(HVC_PMINTENCLR_WRITE, cluster_id, regval);
 }
```


Copyright © 2012 ARM. All rights reserved.
Confidential

3.5 Access to all PMU counters

An additional API call enables configuration/read/write access to all the counters for circumstances where this is preferred. Returned counters belong to processors with the same CPUID on different clusters. For example (on a dual cluster system): if CPU1 uses these API calls, the returned values will be from CPU1 on cluster 0 and CPU1 on cluster1.

3.5.1 Query size of memory to reserve (HVC_GET_COUNTERS_SIZE)

This call permits querying of the ARM Virtualizer software to determine the size of the memory that will need to be reserved for use by subsequent calls.

The illustrative C prototype of the API call is:

unsigned int hvc (HVC_GET_COUNTERS_SIZE);

where:

- HVC_GET_COUNTERS_SIZE is a 32-bit value qualifying the request. See Table 14 for the encoding of this value.
- The HVC will return an unsigned int with the memory size needed for calling HCV_SYNC_PMU_COUNTERS

Table 15: Encoding for HVC_GET_COUNTERS_SIZE

HVC_GET_COUNTERS_SIZE	0x9000_1200

Usage:

```
int array_size;

array_size = hvc (HVC_GET_COUNTERS_SIZE);
```

As per the condensed HVC calling conventions specification:

HVC_GET_COUNTERS_SIZE is passed through register r0.

The return value will be stored in r0.

3.5.2 Synchronize PMU counters (HVC_SYNC_PMU_COUNTERS)

This call permits the configuration, reading and writing of all PMU counters. This is a composite call in that data flow is bi-directional. Some data items are for the consumption of the ARM Virtualizer software while some data is for the consumption of the caller.

The illustrative C API of this call is:

void hvc (HVC_SYNC_PMU_COUNTERS, &descriptor);

where:

- **HVC_SYNC_PMU_COUNTERS** is a 32-bit value qualifying the request (see Table 15 below).
- **&descriptor** is a pointer to caller-allocated memory. The size of this memory is obtained from a prior call with the HVC_GET_COUNTERS_SIZE qualifier. The memory contains objects of type **struct descriptor**.
- struct descriptor is defined as follows:

```
struct descriptor {
    union {
```


Copyright © 2012 ARM. All rights reserved.
Confidential

```
                struct header_ {
                    unsigned int entries; /* INPUT: Number of
                  subsequent objects of type struct counter */
                    unsigned int active_cluster_id; /* OUTPUT:
                  Currently active cluster */
                } header;
                struct counter_{
                    unsigned int cluster_id; /* INPUT/OUTPUT:
                  Cluster ID for the counter in question */
                    unsigned int selected_counter; /* INPUT/OUTPUT:
                  Counter in question */
                    unsigned int event_type; /* INPUT/OUTPUT: Event
                  type of the counter in question */
                    unsigned int counter_value; /* INPUT/OUTPUT:
                  Count value of the counter in question */
                    unsigned int reset_value; /* INPUT: Reload
                  value for the counter in question */
                    unsigned int request_code; /* INPUT: Flags
                  indicating various actions to be performed for the
                  counter in question */
                } counter;
            };
        };
```

The caller-allocated memory must contain a header element of type **struct descriptor**, the **entries** element of which specifies the number of subsequent objects of type **struct descriptor**.

Following a call, the ARM Virtualizer software:

1. Reads the number of valid entries from the header element
2. Fills the **active_cluster_id** field with an encoding of the currently active cluster (See Table 16)
3. Loops from the second element of type struct descriptor onwards until **entries** objects have been processed as follows:

   Read the **request_code** field (see Table 18 for the encoding of this field):

   i. If **request_code** is equal to **PMU_DISABLE_COUNTER** it will disable the counter in question as indicated by the **selected_counter** field (see Table 18 for the encoding of this field).

   ii. If **request_code** is equal to **PMU_CONF_COUNTER** it will program the counter indicated by **selected_counter** to count **event_type** events (see Table 19 for the encoding of this field).

   iii. If **request_code** is equal to **PMU_CONF_RESET_COUNTER** it will program the counter indicated by **selected_counter** to count **event_type** events and then it will update the counter using the value specified in the **reset_value** field. (see Table 19 for the encoding of this field).

   iv. If **request_code** is equal to **PMU_READ_COUNTER** it will fill the **event_type** field and the **counter_value** field with data relevant for the counter indicated by the **selected_counter** field. **event_type** will indicate the event this counter counts and the **counter_value** field will indicate the event count.

Copyright © 2012 ARM. All rights reserved.
Confidential

v. If **request_code** is equal to **PMU_READ_RESET_COUNTER** it will fill the **event_type** field and the **counter_value** with data relevant for the counter indicated by the **selected_counter** and then it will update the counter using the value specified in the **reset_value** field.

Values for counters on an inactive cluster will be written when a cluster switch to that cluster takes place.

The caller is responsible for allocating/deallocating memory for the array.

Usage:

```
unsigned int size;
struct descriptor *descriptor;

size = hvc(HVC_GET_COUNTERS_SIZE);
descriptor = malloc(size);

<prepare descriptor as needed>

hvc(HVC_SYNC_PMU_COUNTERS, descriptor);
```

As per the condensed HVC calling conventions specification:

HVC_SYNC_PMU_COUNTERS is passed through register r0.

The pointer to the descriptor is passed through register r1.

Table 16: Encoding for HVC_SYNC_PMU_COUNTERS

HVC_SYNC_PMU_COUNTERS	0x9000_1201

Table 17: Cluster encoding for use in the active_cluster_id and cluster_id fields

A15_CLUSTER	0
A7_CLUSTER	1

Table 18: PMU counter encodings for use in the selected_counter field

PMU_CYCLE_COUNTER	0x00
PMU_OVERFLOW_FLAG	0x01
PMU_EVENT_COUNTER_N	0x02 + N

Table 19: Encodings for the flags field

PMU_DISABLE_COUNTER	0x01

Copyright © 2012 ARM. All rights reserved.
Confidential

PMU_CONF_COUNTER	0x02
PMU_CONF_RESET_COUNTER	0x03
PMU_READ_COUNTER	0x04
PMU_READ_RESET_COUNTER	0x05

Copyright © 2012 ARM. All rights reserved.
Confidential

4 Debug context migration

When a cluster switch takes place the ARM Virtualizer software saves select CP14 registers on the outbound cluster and will restore them on the inbound cluster. Hardware watchpoint and breakpoint registers are included. This allows self hosted debuggers to continue to function across a switch. External debug interactions using Coresight Debug are not supported.

The following table contains the list of registers which will be saved and restored during a cluster switch:

Table 20: List of saved and restored registers

0x018	DBGWFAR	RW	Watchpoint Fault Address Register, UNK/SBZ
0x01C	DBGVCR	RW	Vector Catch Register
0x024	DBGECR	RW	Event Catch Register
0x080	DBGDTRRX	RW	Host to Target Data Transfer
0x088	DBGDSCR	RW	Debug Status and Control Register
0x08C	DBGDTRTX	RW	Target to Host Transfer
0x094	DBGEACR	RW	Debug External Auxiliary Control Register
0x100-0x114	DBGBVRn	RW	Breakpoint Value Registers
0x140-0x154	DBGBCRn	RW	Breakpoint Control Registers
0x180-0x18C	DBGWVRn	RW	Watchpoint Value Registers
0x1C0-0x1CC	DBGWCRn	RW	Watchpoint Control Registers
0x250-0x254	DBGBXVRn	RW	Breakpoint Extended Value Registers
0x310	DBGPRCR	RW	Device Powerdown and Reset Control Regi
0xF00	DBGITCTRL	RW	Integration Mode Control Register
0xFA0	DBGCLAIMSET	RW	Claim Tag Set Register
0xFA4	DBGCLAIMCLR	RW	Claim Tag Clear Register

4.1 GDB example

The GNU Debugger (GDB) features breakpoint support while debugging. Both the Cortex-A15 and the Cortex-A7 processors support six hardware breakpoints and four watchpoints.

The following operation from the GDB CLI sets a hardware breakpoint on a function_name:

```
$ gdb hbreak <function_name>
```


Copyright © 2012 ARM. All rights reserved.
Confidential

If a cluster switch occurs at a later point, the ARM Virtualizer software ensures that the breakpoint will be valid on the inbound cluster.

Copyright © 2012 ARM. All rights reserved.
Confidential

5 Appendix A: Perf

Here we discuss the implications of the PMU APIs for the Linux perf tool. The focus is on existing perf capability at the time of writing this document.

Linux Perf will use MRC/MCR instructions to access PMU registers. This would be serviced via the Type A interface described earlier.

At present and by design, perf probes the PMUs and configures itself to deal with only the probed PMU type. This poses a problem for a cluster switching system.

Until perf is re-architected to be multi-PMU type aware, perf may be used on a cluster switching system using the ARM Virtualizer software with the caveats exposed in paragraph 3.3

This does mean, however, that in doing so, perf will be ‘mixing’ the event counts as switches occur and this may not always be representative since the different CPU types have vastly different micro-architectures and the same common event will be counted very differently on these.

If you require to use the full functionality of perf on a given processor cluster then it is necessary that:

1. The cluster of interest is the boot cluster
2. Cluster switching is prevented
3. Perf is used on the boot cluster
4. Once done with perf, cluster switching may be enabled.

1. above can be arranged by building the ARM Virtualizer software accordingly.

2. above can be arranged by essentially disabling the cpufreq subsystem. This is possible by changing the governor to the user-space governor and ensuring that no user-space entity is driving cpufreq thereafter. This will ensure that execution remains on the boot cluster. This assumes that a valid cpufreq driver is present on the platform in question and that the driver is the initiator of a cluster switch via the APIs exposed by the ARM Virtualizer software. For details the reader is advised to look at the Linux kernel’s accompanying documentation on cpufreq and governors.

Perf commands:

perf list

This command displays the symbolic event types which can be selected in

the various perf commands with the -e option.

Command list will always return the event list of the boot cluster even if the non-boot cluster is on. Trying to use an architecturally defined counter will work as expected. Using a counter which is not implemented on the non-boot cluster will return 0. The same applies to raw events.

Perf record –e cycles

A processor cycles counter is implemented on both the Cortex-A15 and Cortex-A7 processors. This command will work on both processors and return the correct values if no cluster switch occurs while measuring.

Perf record –e r0C

Copyright © 2012 ARM. All rights reserved.
Confidential

The event 0x0C measures the number of software changes to PC. This hardware counter is present on the Cortex-A7 processor only. This means that Perf will be able to use it only when the boot and active cluster is the Cortex-A7 processor.

If the active cluster is the Cortex-A15 processor this counter will return 0, since the counter is not implemented in hardware.

If the system boots using the Cortex-A15 cluster, this counter will always return 0, even when the Cortex-A7 processor is running, because the counter itself is not common between the Cortex-A15 and Cortex-A7 clusters.

Copyright © 2012 ARM. All rights reserved.
Confidential

6 Appendix B: Streamline

Streamline uses a kernel driver called Gator to collect various statistics.

Gator will use Cortex-A15 and Cortex-A7 processor-specific instructions to access event counters on the specific cluster.

Currently Gator uses the following code to read the PMU Control register:

```
asm volatile("mrc p15, 0, %0, c9, c12, 0" : "=r" (val));
```

On a big.LITTLE system this code will be replaced by appropriate API invocations.

To access the Cortex-A15 PMU Control register :

```
pmcr = hvc (HVC_PMCR_READ, 0);
```

To access the Cortex-A7 PMU Control register:

```
pmcr = hvc (HVC_PMCR_READ, 1);
```

To disable interrupts on the Cortex-A15 processor cluster:

```
hvc (HVC_PMINTENCLR_WRITE, 0, pmintenclr);
```

To disable interrupts on the Cortex-A7 processor cluster:

```
hvc (HVC_PMINTENCLR_WRITE, 1, pmintenclr);
```

The same approach can be used to access other registers described in section 3.1. The API calls use a value in r0 to determine the register and the cluster to access.

If a read is performed on a register belonging to the running cluster, the ARM Virtualizer software will trap and return the current value of the register.

If a write is performed on a register belonging to the running cluster, the ARM Virtualizer software will trap and apply the specified changes immediately to the hardware.

If a read is performed on a register belonging to the inactive cluster, the ARM Virtualizer software will trap and return the old value read from the register and stored before the switch occurred (or 0 if the cluster has never been switched on).

If a write is performed on a register belonging to the inactive cluster, the ARM Virtualizer software will trap and save the operation, which will then be applied when the cluster becomes active at the next cluster switch.

When a cluster switch occurs the ARM Virtualizer software will:

1. Save all the counters for the outbound cluster
2. Save all the PMU control registers on the outbound cluster
3. Restore all the counters for the inbound cluster
4. Restore all the PMU control registers on the inbound cluster

Values from the two clusters will never be mixed. Values saved on a cluster will only be restored on the same cluster. This applies both to PMU control registers and event counters.

Copyright © 2012 ARM. All rights reserved.
Confidential

To synchronize all counters Gator will use the ARM Virtualizer software call described in section 3.2 as follows:

1. Execute HVC_GET_COUNTERS_SIZE to get the size of memory (array_size) needed for subsequent HCV_SYNC_PMU_COUNTERS calls.
2. Allocate memory for the array (malloc(array_size))
3. Fill each element of the array with the informations defining how to program each counter on the system.
    a. To disable a counter:
        i. Fill flag field with PMU_DISABLE_COUNTER
        ii. Fill selected_counter with the macro defining the counter to disable.
    b. To configure a counter:
        i. Fill flag field with PMU_CONF_COUNTER
        ii. Fill selected_counter with the macro defining the counter to configure.
        iii. Fill event_type with the macro defining the event to count.
    c. To read a counter:
        i. Fill flag field with PMU_READ_COUNTER
        ii. Fill selected_counter with the macro defining the counter to read.
    d. To read and reset a counter:
        i. Fill flag field with PMU_READ_RESET_COUNTER
        ii. Fill selected_counter with the macro defining the counter to read and update.
4. Use the appropriate API call to perform the operations on the counters as follows:

    hvc (HVC_SYNC_PMU_COUNTERS, &counters);

5. Deallocate memory as needed.

When the HVC returns, the array will be contain updated values.

Copyright © 2012 ARM. All rights reserved.
Confidential

7 Glossary

The following table describes some of the terms used in this document.

Table 21: Glossary terms

Term	Description
PMU	Performance Monitor Units
HVC	Hypervisor Call

Copyright © 2012 ARM. All rights reserved.
Confidential