Lightweight Profiling Specification

Publication # **43724**	Revision: **3.02**
Issue Date: **October 2007**	

Advanced Micro Devices

Contents

List of Figures

List of Tables

Revision History

Date	Revision	Description
October 2007	3.02	Added IP filters, new events, LWPINS opcode
September 2007	3.01	Minor editorial and formatting changes.
August 2007	3.00	Initial public version.

Chapter 1 Introduction

The lightweight profiling (LWP) proposal extends the AMD64 architecture (in both legacy and long mode) to allow user mode (CPL=3) processes to gather performance data about themselves with very low overhead. Modules such as managed runtime environments and dynamic optimizers can use LWP to monitor the running program with high accuracy and high resolution. They can quickly discover performance problems and opportunities and immediately act on this information.

The proposed extensions allow a program to gather performance data and examine it either by polling or by taking an occasional interrupt. LWP introduces minimal additional states to the CPU and the process. LWP differs from the existing performance counters and from IBS in that large quantities of data are collected before an interrupt is taken. This feature provides a substantial reduction in the overhead of using performance feedback. LWP can even be used with a polling scheme that requires no interrupts at all. LWP also reduces overhead by allowing a user mode program to control its data collection without calling a driver. LWP runs within the context of a thread, so it can be used by multiple processes in a system at the same time.

1.1 Requirements

The following are requirements for LWP to operate properly with modern operating systems (OS):

- **Identifiable—**The OS is able to detect whether LWP is available and, if so, which profiling features are present.
- **Globally Enabled—**The OS must enable LWP in order to allow programs to interact with it. By enabling LWP, the OS commits to context switching the profiling state. By enabling profiling interrupts, the OS commits to handling them.
- **Secure—**No data on the operation of the OS may “leak” to any user process. No data on the operation of one user process may leak to any other process.
- **Separable—**The hardware mechanisms for LWP do not interact in any way with the existing performance counters or instruction-based sampling.

1.2 Overview

When enabled, LWP hardware monitors one or more events during the execution of user-mode code and periodically inserts event records into a ring buffer in the address space of the running process. When the buffer is filled beyond a user-specified threshold, the hardware can cause an interrupt which the OS can use to signal a process to empty the buffer. With proper OS support, the interrupt can even be delivered to a separate process or thread.

Instructions are only counted if they execute in user mode (CPL=3) and contribute to the instruction count in that mode according to AMD standard for counting instructions. Furthermore, LWP is inactive while in system management mode (SMM) or while entering or leaving SMM.

Once LWP is enabled, the user thread has complete control over its operations via the *LLWPCB* and *SLWPCB* instructions. Each thread in a multi-threaded process must configure LWP separately. A thread has its own buffer and counters which are context switched with the rest of the thread state. However, it is certainly possible for a single monitor thread to collect, store, and process the data from multiple other threads in the process.

During profiling, the LWP hardware monitors and reports on one or more types of events. Following are the steps in this process:

1. **Count—**Each time an instruction is retired, LWP decrements its internal event counters for all of the events associated with the instruction. An instruction can cause zero, one, or multiple events. For instance, an indirect jump through a pointer in memory counts as an instruction retired, a branch retired, and may also cause up to two DCache misses (or more, if there is a TLB miss) and up to two ICache misses.

    - Some events may have filters or conditions on them that regulate counting. For instance, the user may configure LWP so that only cache miss events with latency greater than a specified minimum are eligible to be counted.

2. **Gather—**When an event counter reaches zero, the event should be reported, and LWP gathers an event record. This is the equivalent of filling in an internal copy of an event record, though actual implementation may vary. The event's counter freezes at zero until an event record is written to the event buffer.

    For most events, such as instructions retired, LWP gathers an event record describing the instruction that caused the counter to reach zero. However, it is valid for LWP to gather event record data for the *next* instruction that causes the event, or to take other alternatives to capture the record. Some of these options are described with the individual events.

    - An implementation can choose to gather event information on one or many events at any one time. If multiple event counters reach zero, an advanced LWP implementation may gather one event record per event and write them sequentially. A basic LWP implementation may choose one of the eligible events. The other expired events wait until the chosen event record is written and then pick the next eligible instruction for the waiting event. This situation should be extremely uncommon if software chooses its intervals to be large enough.
    - LWP may discard an event occurrence. For instance, if the event buffer needs to be paged in from disk, LWP might not be able to preserve the pending event record data. If an event is discarded, LWP gathers an event record for the next instruction to cause the event.
    - Similarly, if LWP needs to replay an instruction to gather a complete event record, the replay may abort instead of retiring. The event counter remains zero and LWP gathers an event record for the next instruction to cause the event.

3. **Store**—When a complete event record is gathered, LWP stores it into a ring buffer in the process' address space and advances the ring buffer pointer.
    - If the ring buffer is full at this time, LWP increments a 64-bit counter of missed events and does not advance the buffer pointer.
    - If more than one event record reaches the Store stage simultaneously, only one need be stored. LWP may delay storing other event records or it may discard the information and proceed to choose the next eligible instruction for the discarded event type(s).
    - The store need not complete synchronously with the instruction retiring. In other words, if LWP buffers the event record contents, the Store stage (and subsequent stages) may complete some number of cycles after the tagged instruction retires. The data about the event and the instruction are precise, but the rest of the LWP process may complete later.
4. **Report**—If LWP threshold interrupts are enabled and the space used in the ring buffer exceeds a user-defined threshold, LWP initiates an interrupt. The OS can use this to signal the process to empty the buffer. Note that the interrupt may occur significantly later than the event that caused the threshold to be met.
5. **Reset**—The counter for the event that was stored is reset to its programmed interval (with any randomization applied). Counting for that event starts again. Reset happens if the event record is stored or if the missed event counter was incremented.

The user process can wait until an interrupt occurs to process the events in the ring buffer. This requires OS or driver support. (As a consequence, interrupts can only be enabled if a kernel mode routine allows it; refer to "LWP Model Specific Registers" on page 20.) One usage model is to have the program call a driver to associate the LWP interrupt with a semaphore or mutex. When the interrupt occurs, the driver signals the associated object. Any thread waiting on the object will wake up and can process the buffer. (Other driver models are possible, of course.)

Alternatively, the user process can have a thread that periodically polls the ring buffer and removes event records from it, advancing the tail pointer so that the LWP hardware can continue storing records. The hardware is designed to never overflow the buffer by advancing the head pointer to equal the tail pointer.

1.3 Events and Event Records

When a monitored event overflows its event counter, LWP puts an event record into the LWP event ring buffer. Each event record in the ring buffer is 32 bytes long in version 1 of LWP. (The actual event record size is returned as *LWPEventSize* by the *CPUID instruction* when querying for LWP features.)

Reserved fields and fields that are not defined for the particular event are set to zero when LWP writes an event record.

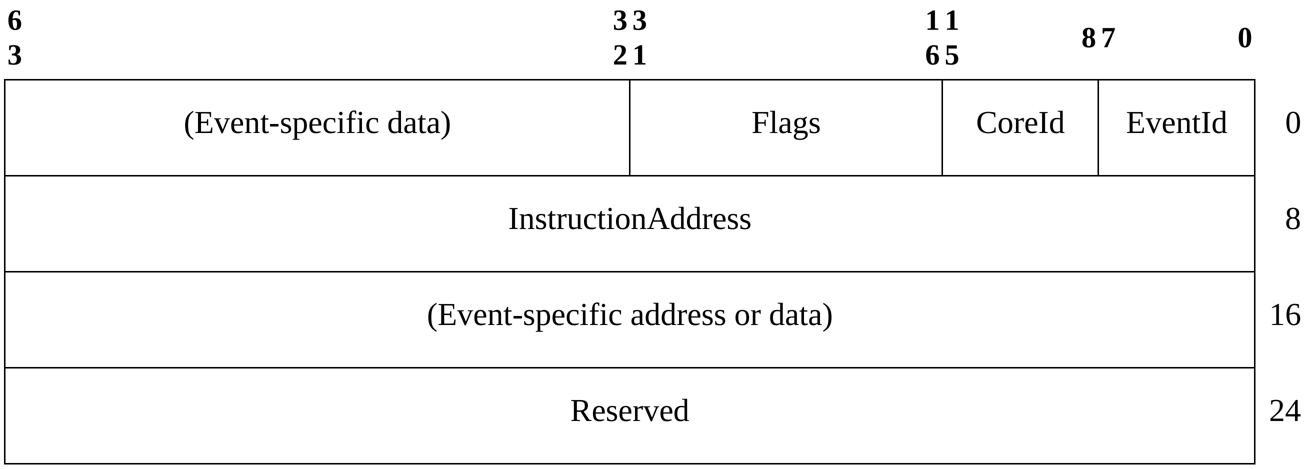

Figure 1. Generic Event Record

Table 1. Generic Event Record Fields

Field	Bytes	Bits	Description
EventId	0		Event identifier specifying the event record type. Valid identifiers are 1 to 255. 0 is an invalid identifier.
CoreId	1		CPU core number. For multicore systems, this identifies the core on which LWP is running. This allows software to aggregate event records from multiple threads into a single buffer without losing CPU information. 0 for single core systems.
Flags	2:3	16:31	Event-specific flags. Flags are typically allocated starting at bit 31.
	4:7		Event-specific data.
InstructionAddress	8:15		Linear address of the instruction that triggered this event record. This is the value after adding in the CS base address. If the base is non-zero, software must track it. (Generally, modern operating systems use a CS base of zero.)
	16:23		Event-specific address or other data.
	24:31		Reserved.

Table 2 lists the event identifiers for the events available in version 1 of LWP. They are described in detail in the following sections.

Table 2. EventId Values

EventId	Description
0	Reserved – invalid event
1	Instructions retired
2	Branches retired
3	DCache misses
4	CPU clocks not halted
5	CPU reference clocks not halted

1.3.1 Instructions Retired

LWP decrements the event counter each time an instruction retires. When the counter reaches zero, it stores a generic event record with an EventId of 1.

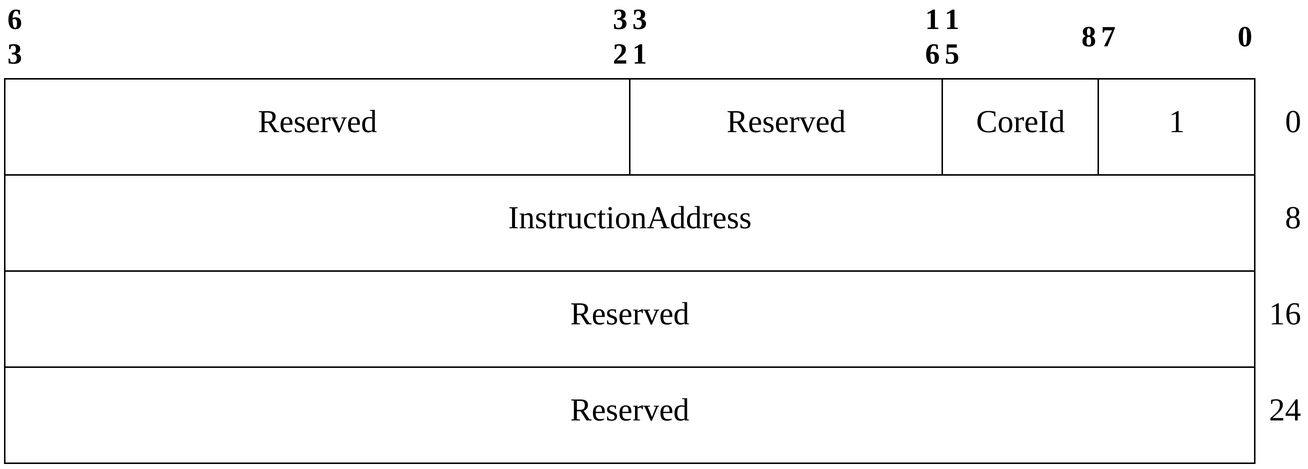

Figure 2. Instruction Retired Event Record

1.3.2 Branches Retired

LWP decrements the event counter each time a transfer of control retires, regardless of whether or not it is taken. When the counter reaches zero, it stores an event record with an EventId of 2.

Control transfer instructions are short and long jumps (including JCXZ and its variants), LOOPx, CALL, and RET. LWP does not count traps or interrupts, whether synchronous or asynchronous, nor does it count operations that switch to or from ring 3, SMM, or SVM, such as SYSCALL, SYSENTER, or INT 3.

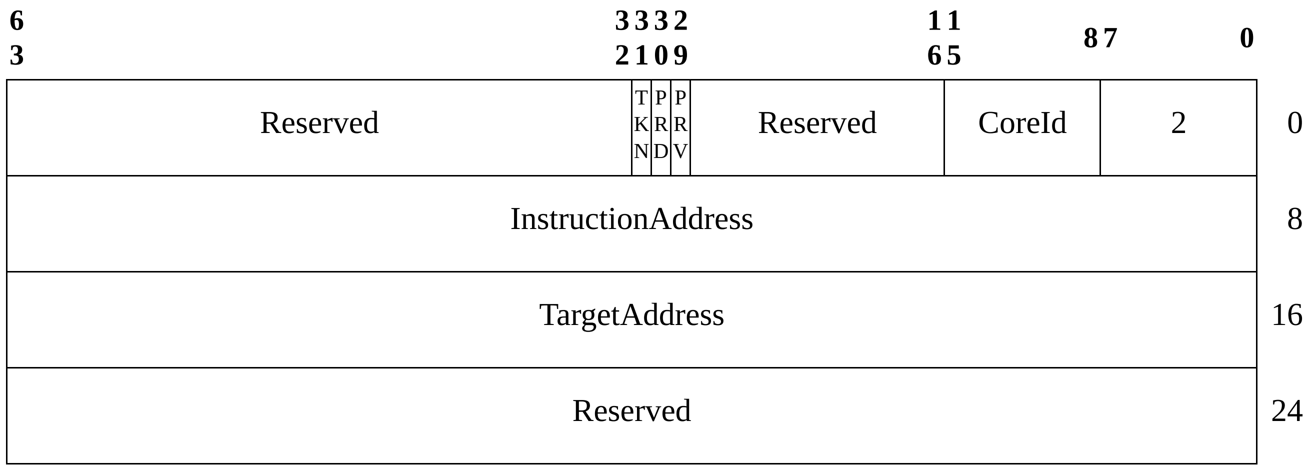

Figure 3. Branch Retired Event Record

Table 3. Branch Retired Event Record Fields

Field	Bytes	Bits	Description
EventId	0		Event identifier = 2
CoreId	1		CPU core number
	2:3	16:29	Reserved
PRV	3	29	1—PRD bit is valid 0—Prediction information is not available Some implementations of LWP may be unable to capture branch prediction information on some or all branches.
PRD	3	30	1—Branch was predicted correctly 0—Mispredicted Always 1 for unconditional direct branches.
TKN	3	31	1—Branch was taken 0—Branch not taken Always 1 for unconditional branches.
	4:7		Reserved
InstructionAddress	8:15		Instruction address
TargetAddress	16:23		Address of instruction after branch. This is the target if the branch was taken and the fall-through address if the branch was a not-taken conditional branch.
	24:31		Reserved

1.3.3 DCache Misses

LWP decrements the event counter each time a load from memory causes a DCache miss whose latency exceeds the *LWPCacheLatency* threshold and/or whose data comes from a level of the cache or memory hierarchy that is selected for counting. When the counter reaches zero, LWP stores an event record with an EventId of 3.

A misaligned access that causes two misses on a single load only decrements the event counter by 1 and, if it reports an event, the data is for the lowest address that missed. LWP does not count cache misses that are indirectly due to TLB walks, LDT or GDT references, TLB misses, etc. It only counts loads directly caused by the instruction. Cache misses caused by the LWP hardware itself are not subject to counting.

1.3.3.1 Measuring Latency

The x86 architecture allows multiple loads to be outstanding simultaneously. An implementation of LWP might not have a full latency counter for every load that is waiting for a cache miss to be resolved. Therefore, an implementation may apply any of the following simplifications. Software using LWP should be prepared for this.

- The implementation may round the latency to a multiple of 2^*j*. This is a small power of 2, and the value of *j* must be 1 to 4. For example, in the rest of this section, assume that *j* = 4, so 2^*j* = 16. The low 4 bits of latency reported in the event record will be 0. The actual latency counter is incremented by 16 every 16 cycles of waiting. The value of *j* is returned as *LWPLatencyRnd* by the *CPUID instruction* when querying for LWP features.
- The implementation may do an approximation when starting to count latency. If counting is in increments of 16, the 16 cycles need not start when the load begins to wait. The implementation may bump the latency value from 0 to 16 any time during the first 16 cycles of waiting.
- The implementation may cap total latency to 2^*n*-16 (where *n* >= 10). The latency counter is thus a saturating counter that stops counting when it reaches its maximum value. For example, if *n* = 10, the latency value will count from 0 to 1008 in steps of 16 and then stop at 1008. (If *n* = 10, each counter can be a mere 6 bits wide.) The value of *n* is returned as *LWPLatencyMax* by the *CPUID instruction* when querying for LWP features.

Note also that the latency threshold used to filter events is a multiple of 16 when performing the comparison that decides whether a cache miss event is eligible to be counted.

1.3.3.2 Reporting the DCache Miss Data Address

The event record for a DCache miss reports the linear address of the data. The way an implementation records the linear address affects the exact event that is reported and the amount of time it takes to report a cache miss event. The implementation may report the event immediately, report the next eligible event once the counter reaches zero, or replay the instruction.

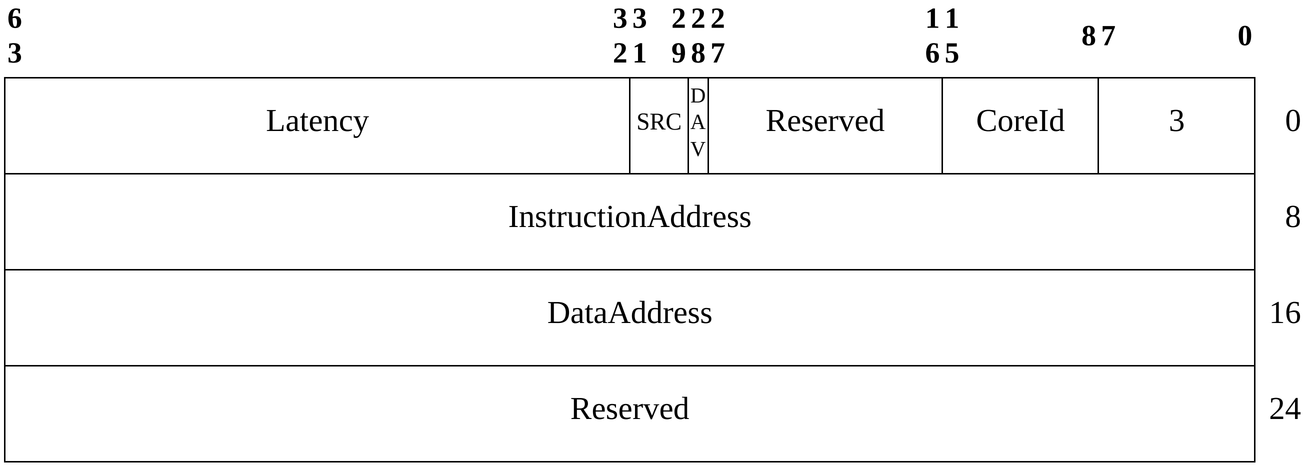

Figure 4. DCache Miss Event Record

Table 4. DCache Miss Event Record Fields

Field	Bytes	Bits	Description
EventId	0		Event identifier = 3
CoreId	1		CPU core number
	2:3	16:27	Reserved
DAV	3	28	1—DataAddress is valid 0—Address is unavailable
SRC	3	29:31	Data source for the requested data 0 No valid status 1 Local L3 cache 2 Remote CPU or L3 cache 3 DRAM 4 Reserved (for Remote cache) 5 Reserved 6 Reserved 7 Other (MMIO/Config/PCI/APIC)
Latency	4:7		Total latency of cache miss (in cycles)
InstructionAddress	8:15		Instruction address

Table 4. DCache Miss Event Record Fields (Continued)

Field	Bytes	Bits	Description
DataAddress	16:23		Address of memory reference (if flag bit 28 = 1)
	24:31		Reserved

1.3.4 CPU Clocks not Halted

LWP decrements the event counter each clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter reaches zero, it stores a generic event record with an EventId of 4. This counter varies in real-time frequency as the core clock frequency changes.

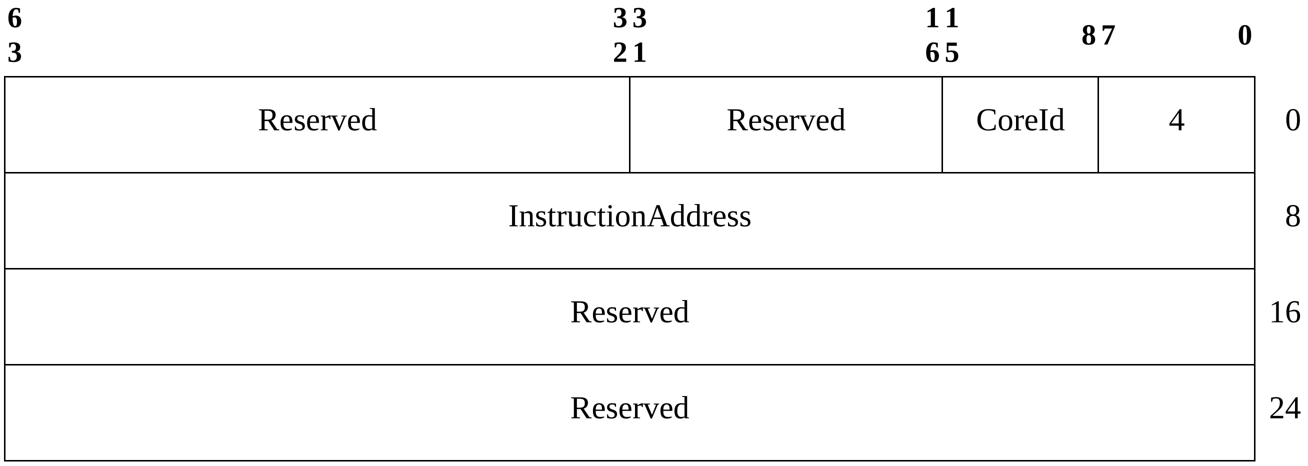

Figure 5. CPU Clocks not Halted Event Record

1.3.5 CPU Reference Clocks not Halted

LWP decrements the event counter each reference clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter reaches zero, it stores a generic event record with an EventId of 5.

The reference clock runs at a constant frequency that is independent of the core frequency and of the performance state. The reference clock frequency is processor dependent. The processor may implement this event by subtracting the ratio of (reference clock frequency / core clock frequency) each core clock cycle.

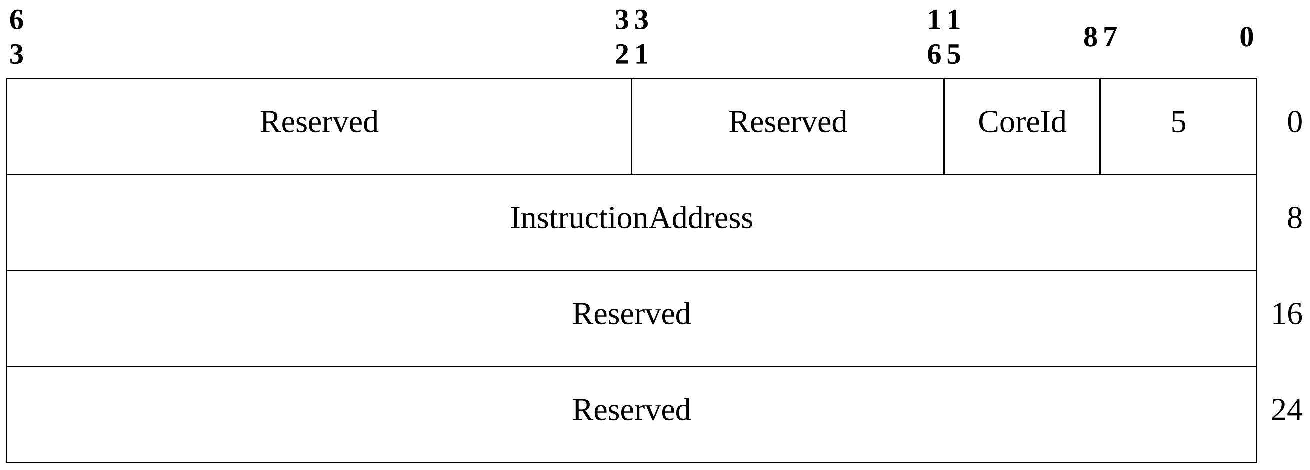

Figure 6. CPU Reference Clocks not Halted Event Record

1.3.6 Programmed Event

When a program successfully executes the LWPINS instruction (see “LWPINS—Insert User Event Record into LWP Event Ring Buffer” on page 23), the processor stores an event record with an event identifier of 255.

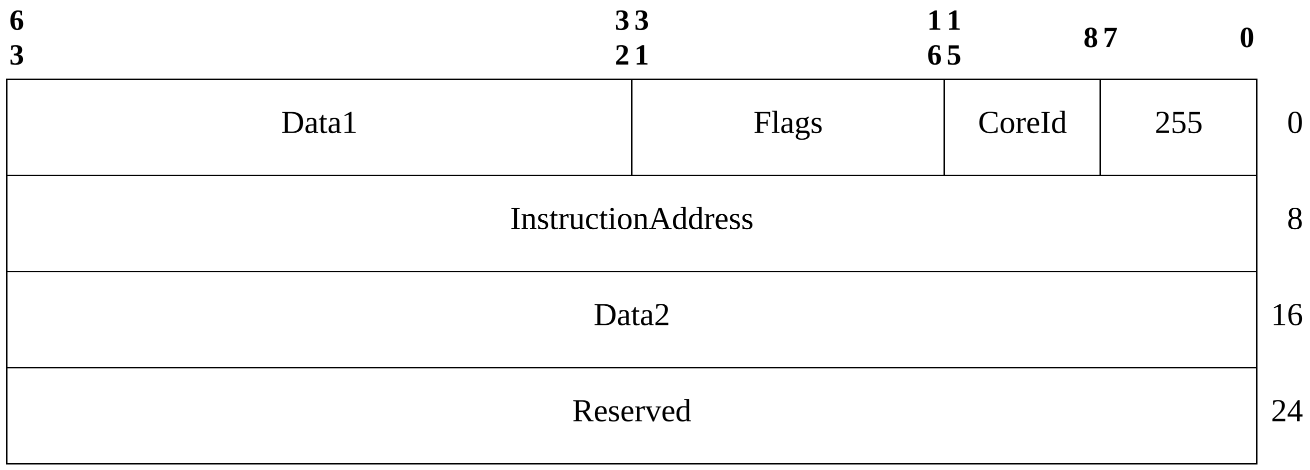

Figure 7. Programmed Event Record

Table 5. Programmed Event Record Fields

Field	Bytes	Bits	Description
EventId	0		Event identifier = 255
CoreId	1		CPU core number

Table 5. Programmed Event Record Fields (Continued)

Field	Bytes	Bits	Description
Flags	2:3		Flags value from EAX[15:0]
Data1	4:7		Data value from EBX
InstructionAddress	8:15		Instruction address
Data2	16:23		Data value from rDX, zero extended if running in legacy mode
	24:31		Reserved

1.3.7 Other Events

The overall design of LWP allows easy extension to the list of events that it can monitor. The following are possibilities for events that may be added in future versions of LWP:

- DTLB misses
- FPU operations
- ICache misses
- ITLB misses

Chapter 2 LWP Details

This chapter describes how to identify the presence of lightweight profiling by using the CPUID instruction and provides information on LWP MSRs, instructions, and other considerations.

2.1 CPUID Identification

To identify whether lightweight profiling is present, use the CPUID instruction:

- Call: CPUID <= EAX: 8000_0001
- Return: EDX bit ***TBD*** is set to 1 if LWP is present.

To identify the supported LWP capabilities, use CPUID with the following leaf request code:

- Call: CPUID <= EAX: 8000_001C (for extended features, EAX: 8000_001D)
- Return: See Table 6, "Lightweight Profiling CPUID Values"

The bits returned in EAX are taken from LWPMSR0 and reflect the currently enabled LWP features. These are a subset of the bits returned in EDX, which reflect the full capabilities of LWP in the current processor. The operating system can enable a subset of LWP if it does not support all available features. For instance, if the OS cannot handle an LWP threshold interrupt, it can disable the feature. User-mode software must assume that the bits in EAX describe the features it can use. Operating systems should use the bits from EDX to determine the capabilities of LWP and enable all or some of the available features.

Under SVM, if a VMM allows the migration of guests among processors that all have LWP available, it must arrange for CPUID to report the logical AND of the available feature bits and the minimum of the number of events over all processors in the migration set.

Table 6. Lightweight Profiling CPUID Values

Field	Reg	Bits	Description
Enabled	EAX	0	LWP is enabled. If this bit is 0, the remainder of the data returned by CPUID should be ignored.
IRE	EAX	1	Instructions retired event (EventId = 1) is enabled.
BRE	EAX	2	Branch retired event (EventId = 2) is enabled.
DME	EAX	3	DCache miss event (EventId = 3) is enabled.
CNH	EAX	4	CPU clocks not halted event (EventId = 4) is enabled.
RNH	EAX	5	CPU reference clocks not halted event (EventId = 5) is enabled.
	EAX	6:29	Reserved for future events.

Table 6. Lightweight Profiling CPUID Values (Continued)

Field	Reg	Bits	Description
Extension	EAX	30	Extended CPUID information is available. If 1, information on events with EventId > 29 is available by executing CPUID with EAX = 8000_001D. (0 for LWP Version 1.)
Interrupt	EAX	31	Interrupt on threshold overflow is enabled.
LWPCBSize	EBX	7:0	Size in bytes of the LWPCB. This value is at least LWPEventOffset + (LWPMaxEvents * 8) but an implementation may require a larger control block.
LWPEventSize	EBX	15:8	Size in bytes of an event record in the LWP ring buffer. (32 for LWP Version 1.)
LWPMaxEvents	EBX	23:16	Number of different events that can be monitored simultaneously.
LWPEventOffset	EBX	31:24	Offset from the start of the LWPCB to the EventInterval0 field. Software must use this value to locate the area of the LWPCB that describes events to be sampled. This permits expansion of the initial fixed region of the LWPCB.
LWPLatencyMax	ECX	4:0	Number of bits in cache latency counters (10 to 31).
LWPDataAddress	ECX	5	1—Cache miss event records report the data address of the reference. 0—Data address is not reported.
LWPLatencyRnd	ECX	8:6	The amount by which cache latency is rounded. The bottom LWPLatencyRnd bits of latency information will be zero. The actual number of bits implemented for the counter is LWPLatencyMax – LWPLatencyRnd. Must be 0 to 4.
LWPVersion	ECX	15:9	Version of LWP implementation. (1 for LWP Version 1.)
	ECX	29:16	Reserved
LWPCacheLevels	ECX	30	1—Cache-related events can be filtered by the cache level that returned the data. The value of CLF in the LWPCB enables cache level filtering. 0—CLF is ignored An implementation must support filtering either by latency or by cache level. It may support both.
LWPCacheLatency	ECX	31	1—Cache-related events can be filtered by latency. The value of MinLatency in the LWPCB controls filtering. 0—MinLatency is ignored An implementation must support filtering either by latency or by cache level. It may support both.
Enabled	EDX	0	LWP is available. If this bit is 0, the remainder of the data returned by CPUID should be ignored.

Table 6. Lightweight Profiling CPUID Values (Continued)

Field	Reg	Bits	Description
IRE	EDX	1	Instructions retired event (EventId = 1) is available.
BRE	EDX	2	Branch retired event (EventId = 2) is available.
DME	EDX	3	DCache miss event (EventId = 3) is available.
CNH	EAX	4	CPU clocks not halted event (EventId = 4) is available.
RNH	EAX	5	CPU reference clocks not halted event (EventId = 5) is available.
	EDX	6:30	Reserved for future events.
Interrupt	EDX	31	Interrupt on threshold overflow is available.

2.2 LWP Model Specific Registers

The LWP model-specific registers describe and control the LWP hardware. They are available if EDX bit ***TBD*** of CPUID 8000_0001 is 1.

2.2.1 LWPMSR0—LWP Feature Enable

LWPMSR0 controls how LWP can be used on the processor. It can prohibit the use of LWP or restrict it in several ways. The operating system loads LWPMSR0 at start-up time (or at the time a LWP driver is loaded) to indicate its level of support for LWP. Only bits that are set in EDX from CPUID when enumerating LWP can be turned on in LWPMSR0. Attempting to set other bits causes a #GP fault.

User code can examine the contents of LWPMSR0 by executing CPUID with EAX = 8000_001C and then examining the contents of EAX.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INT	Reserved																									RNH	CNH	DME	BRE	IRE	EN

Figure 8. LWPMSR0—Lightweight Profiling Feature Enable

Table 7. LWPMSR0—Lightweight Profiling Feature Enable Fields

Field	Bits	Description
EN	0	Enable LWP.
IRE	1	Allow LWP to count instructions retired.

Table 7. LWPMSR0—Lightweight Profiling Feature Enable Fields (Continued)

Field	Bits	Description
BRE	2	Allow LWP to count branches retired.
DME	3	Allow LWP to count DCache misses.
CNH	4	Allow LWP to count CPU clocks not halted.
RNH	5	Allow LWP to count CPU reference clocks not halted.
	6:30	Reserved
INT	31	Allow LWP to generate an interrupt when threshold is exceeded.

2.2.2 LWPMSR1—LWPCB Pointer

LWPMSR1 provides access to the internal copy of the LWPCB pointer. RDMSR on this register performs the operations described for the SLWPCB instruction, and WRMSR performs the LLWPCB operations. Note that RDMSR and WRMSR are only available in ring 0, while the LLWPCB and SLWPCB instructions are available in user mode.

2.2.3 LWPMSR2—LWP Core ID

LWPMSR2 contains the value that will be stored into the CoreId field of every event record written by LWP on this processor. The operating system should initialize this value to be the local APIC number obtained by executing CPUID function 0000_0001.

This MSR is present so that when LWP is runs in a virtualized environment, it has access to the core number without needing to enter the hypervisor.

2.3 LWP Control Instructions

The LLWPCB instruction enables and disables lightweight profiling and controls the events being profiled. The SLWPCB instruction queries the current state of lightweight profiling. These instructions provide user mode access to the LWPCB pointer in LWPMSR1.

LLWPCB—Load LWPCB Pointer

Sets the state of the lightweight profiling hardware from the LWP Control Block at DS:rAX and enables profiling if specified. Returns the previous value of the LWPCB address in rAX.

The LWPCB must be aligned on a 16-byte boundary in normal (writeback) memory and must be writable in user mode. Software is advised to place the LWPCB so that it does not cross a page boundary, but this is not a requirement. To disable LWP, execute LLWPCB with rAX = 0.

This operation may only be issued when the machine is in protected mode. It can be executed at any privilege level. If it is executed from a privilege level other than 3, the internal LWPCB pointer is loaded, but the initialization of the remainder of LWP state is deferred until the processor enters ring 3. This allows the LWPCB to reside in memory that needs to be paged in immediately; the page fault will occur from ring 3.

If LWP is currently active, it flushes its internal state to memory in the old LWPCB. Then it sets up new internal state from the new LWPCB and writes the new LWPCB.Flags bits to indicate the resulting status of LWP. These bits indicate which events are actually being profiled and whether threshold interrupts are enabled. Bits [1:30] correspond to events with EventId of the same value.

If no events are being collected, the flags are set to zero and LWP is disabled. In this case, a subsequent SLWPCB will return zero in rAX. This can happen if none of the EventId fields in the LWPCB select events that are implemented and enabled on the current system.

If multiple event selectors specify the same EventId, only the earliest one in the LWPCB is used. LWP ignores duplicate events and treats them as if the EventId were zero, though it does not change the EventId field in the LWPCB.

Since the previous LWPCB address is returned in rAX, a program can temporarily disable LWP by executing LLWPCB with rAX = 0 and saving the old LWPCB address returned in rAX. It can later re-enable LWP by executing LLWPCB with rAX set to the old address.

rFLAGS Affected

None

Exceptions

Invalid opcode, #UD—The LLWPCB instruction is not supported, or LWP is not implemented on this processor, or profiling is not enabled in LWPMSR0, or the system is not in protected mode.

Page Fault, #PF—The memory at [DS:rAX : DS:rAX + LWPCBSize - 1] is not writeable by the current process.

SLWPCB—Store LWPCB Pointer

Flushes the current state of LWP into the LWPCB in memory and returns the current address of the LWPCB in rAX. If LWP is not currently active, SLWPCB sets rAX to zero.

This operation may only be issued when the machine is in protected mode. It can be executed at any privilege level.

rFLAGS Affected

None

Exceptions

Invalid opcode, #UD—The SLWPCB instruction is not supported, or LWP is not implemented on this processor, or profiling is not enabled in LWPMSR0, or the system is not in protected mode.

LWPINS—Insert User Event Record into LWP Event Ring Buffer

Inserts a record into the LWP event ring buffer in memory and advances the ring buffer pointer. The record has an EventId of 255. The Flags bits are taken from EAX[15:0]. The event-specific data at bytes 7:4 of the event record are taken from EBX. The event-specific data at bytes 23:16 are taken from rDX and are zero extended if the program is running in legacy mode. See Figure 7, “Programmed Event Record” for details.

If the record is successfully stored, the CF flag is cleared. If the ring buffer is full and the record cannot be stored, the CF flag is set.

This operation may only be issued when the machine is in protected mode. It can be executed at any privilege level.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. Such events might include information on changes in the process’ address space such as library loads and unloads, or changes in the execution environment such as a change in the state of a user-mode thread of control.

rFLAGS Affected

CF

Exceptions

Invalid opcode, #UD—The LWPINS instruction is not supported, or LWP is not implemented on this processor, or profiling is not enabled in LWPMSR0, or LWP is not running (LWPMSR1 is zero), or the system is not in protected mode.

2.4 LWP Control Block

The LWP Control Block (LWPCB) specifies the details of how LWP operates. It is an interactive region of memory in which some fields are controlled and modified by the LWP hardware and others are controlled and modified by the software that processes the LWP event records.

Most of the fields in the LWPCB are constant for the duration of a LWP session (the time between enabling LWP and disabling it). This means that they are loaded into the LWP hardware when it is enabled, and may be periodically reloaded from the same location as needed. The contents of the constant fields must not be changed during a LWP run or results will be unpredictable. Changing the LWPCB memory to read-only or unmapped will cause an exception the next time the LWP hardware attempts to access it. To change values in the LWPCB, disable LWP, change the LWPCB (or create a new one), and reenable LWP.

A few fields are modified by the LWP hardware to communicate progress to the software that is emptying the event record buffer. Software may read them but should never modify them during an LWP session. Other fields are for software to modify to indicate that progress has been made in emptying the buffer. Software writes these fields and the LWP hardware will read them as needed.

For efficiency, some of the LWPCB fields may be shadowed in registers in the LWP hardware unit when profiling is active. LWP will refresh these fields from (or flush them to) memory as needed to allow software to make progress. For more information, refer to "LWPCB Access" on page 31.

All fields in the LWPCB (as shown in Figure 9) that are marked as "Reserved" must be zero.

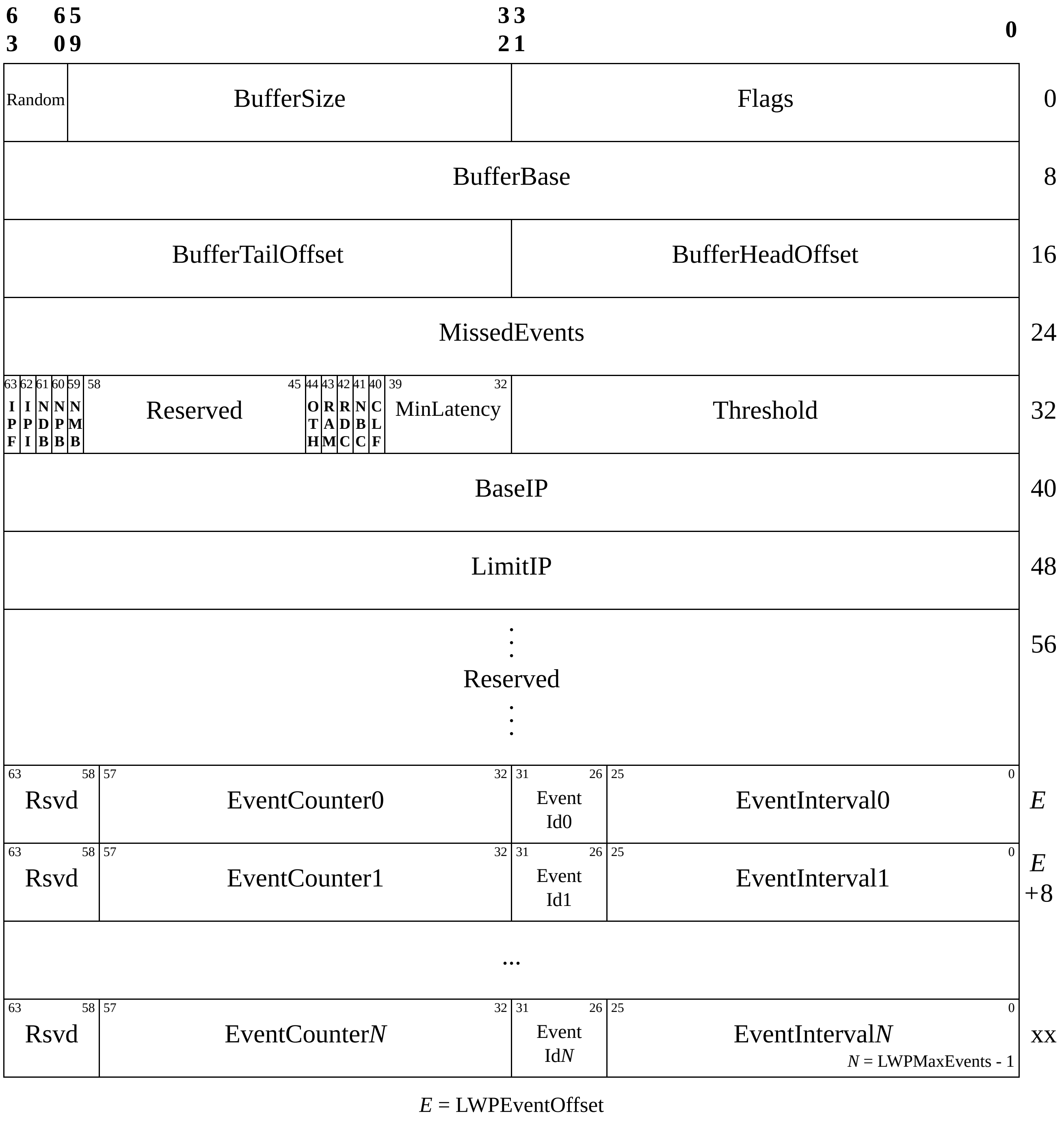

Figure 9. LWPCB—Lightweight Profiling Control Block

The R/W column in Table 8, “LWPCB—Lightweight Profiling Control Block Fields” indicates how a field is used while LWP is enabled:

- LWP—hardware modifies the field
- Init—hardware modifies the field while executing LLWPCB
- SW—software may modify the field
- No—field must remain unchanged as long as the LWPCB is in use

Table 8. LWPCB—Lightweight Profiling Control Block Fields

Field	Bytes	Bits	Description	R/W
Flags	3:0		Flags indicating LWP state (see Table 9, “LWP Flags”).	Init
BufferSize	7:4	59:32	Total size of the event record buffer (in bytes). Must be a multiple of the event record size *LWPEventSize*.	No
Random	7	63:60	Number of bits of randomness to use in counters. Each time a counter is loaded from an interval to start counting down to the next event to record, the bottom Random bits are set to a random value. This avoids fixed patterns in events.	No
BufferBase	15:8		Linear address of the event record buffer. Must be aligned on a 32-byte boundary (the low 5 bits of BufferBase are ignored). Software is encouraged to align the buffer on a page boundary, but this is not required.	No
BufferHeadOffset	19:16		Unsigned offset into BufferBase specifying where the LWP hardware will store the next event record. When BufferHeadOffset == BufferTailOffset, the buffer is empty. BufferHeadOffset is always less than BufferSize and is always a multiple of *LWPEventSize*.	LWP
BufferTailOffset	23:20		Unsigned offset into BufferBase specifying the oldest event record in the buffer. BufferTailOffset is always less than BufferSize and is always a multiple of *LWPEventSize*.	SW
MissedEvents	31:24		The 64-bit count of the number of events that were missed. A missed event occurs after LWP stores an event record when it advances BufferHeadOffset and discovers that it would be equal to BufferTailOffset. In this case, LWP leaves BufferHeadOffset unchanged and instead increments the MissedEvents counter. Thus, when the buffer is full, the last event record is overwritten.	LWP

Table 8. LWPCB—Lightweight Profiling Control Block Fields (Continued)

Field	Bytes	Bits	Description	R/W
Threshold	35:32		If non-zero, threshold for interrupting the user to indicate that the buffer is filling up. When LWP advances BufferHeadOffset, it computes the space used as ((BufferHeadOffset – BufferTailOffset) % BufferSize) and compares it to Threshold. If the space used == Threshold and threshold interrupts are enabled, LWP causes an interrupt. The compare for equality ensures that only one interrupt occurs when the threshold is crossed. **Notes:** If zero, no threshold interrupts will be generated. This field is ignored if threshold interrupts are not enabled in LWPMSR1.	No
MinLatency	36		Minimum latency required to make a cache-related event eligible for LWP counting. Applies to all cache-related events being monitored. The number in MinLatency is multiplied by 16 to get the actual latency in cycles. This scaling provides less resolution but a larger range for filtering. An implementation may have a maximum for the latency value it captures. If MinLatency*16 exceeds this maximum value, the maximum is used instead. A value of 0 disables filtering by latency. ***Notes:*** MinLatency is ignored if no cache latency event is chosen in one of the EventId*n* fields. MinLatency is ignored if *CPUID* indicates that the implementation does not filter by latency. Use the CLF bits to get a similar effect. At least one of these mechanisms must be available.	No
CLF	37	40	Cache Level Filtering. 1—Enables filtering cache-related events by the cache level or memory level that returned the data. It enables the next 4 bits, and cache-related events are only eligible for LWP counting if the bit describing the memory level is on. 0—Disables cache level filtering. The next 4 bits are ignored, and any cache or memory level is eligible. ***Notes:*** CLF is ignored if no cache latency event is chosen in one of the EventId*n* fields. CLF is ignored if *CPUID* indicates that the implementation does not filter by cache level. Use the MinLatency field to get a similar effect. At least one of these mechanisms must be available.	No

Table 8. LWPCB—Lightweight Profiling Control Block Fields (Continued)

Field	Bytes	Bits	Description	R/W
NBC	37	41	Set to 1 to record cache-related events that are satisfied from data held in a cache that resides on the Northbridge. Ignored if CLF is 0.	No
RDC	37	42	Set to 1 to record cache-related events that are satisfied from data held in a remote data cache. Ignored if CLF is 0	No
RAM	37	43	Set to 1 to record cache-related events that are satisfied from DRAM. Ignored if CLF is 0	No
OTH	37	44	Set to 1 to record cache-related events that are satisfied from other sources, such as MMIO, Config space, PCI space, or APIC. Ignored if CLF is 0	No
NMB	39	59	No mispredicted branches. 1—Mispredicted branches will not be counted. 0—Mispredicted branches will be counted if not suppressed by other filter conditions. Caution: If NMB and NPB are both set, no branches will be counted. This value is ignored if the *Branches Retired* event is not chosen in one of the EventId*n* fields.	No
NPB	39	60	No predicted branches. 1—Correctly predicted branches will not be counted. 0—Correctly predicted branches will be counted if not suppressed by other filter conditions. Caution: If NMB and NPB are both set, no branches will be counted. This value is ignored if the *Branches Retired* event is not chosen in one of the EventId*n* fields.	No
NDB	39	61	No direct branches. 1—Direct branches will not be counted. This only applies to unconditional RIP-relative branches. Conditional branches, indirect jumps through a register or memory, calls, and returns are counted normally. 0—Direct branches will be counted if not suppressed by other filter conditions. This value is ignored if the *Branches Retired* event is not chosen in one of the EventId*n* fields.	No
IPI	39	62	1—IP filtering inverted. Instructions outside the range from BaseIP to LimitIP are eligible for LWP counting. 0—IP filtering normal. Instructions inside the range from BaseIP to LimitIP are eligible for LWP counting. Ignored if IPF is zero.	No

Table 8. LWPCB—Lightweight Profiling Control Block Fields (Continued)

Field	Bytes	Bits	Description	R/W
IPF	39	63	1—IP filtering enabled. The values of the BaseIP and LimitIP fields specify a range of instruction addresses that are eligible for LWP event counting and reporting. The range is inclusive if IPI is 0 and exclusive if IPI is 1. 0—IP filtering disabled. BaseIP, LimitIP, and IPI are ignored; instructions at every address are eligible for LWP counting.	No
BaseIP	47:40		Low limit of the eligible IP filtering range. An instruction must be at a location greater than or equal to BaseIP to be in the range. Ignored if IPF is zero.	No
LimitIP	55:48		High limit of the eligible IP filtering range. An instruction must be at a location less than or equal to LimitIP to be in the range. Ignored if IPF is zero.	No
	E-1:56		Reserved area between the fixed portion of the LWPCB and the event specifiers. Must be zero. The EventInterval0 field is at offset *E* = *LWPEventOffset* which is returned by *CPUID*.	
EventInterval0	*E*+3:*E*	25:0	Number of events of type EventId0 to count before storing an event record.	No
EventId0	*E*+3	31:26	EventId of the event to count in this counter. 0 means disable this counter. It is invalid to specify the same EventId in two or more counters and may cause unpredictable results.	No
EventCounter0	*E*+7: *E*+4	57:32	Starting or current value of counter	LWP
Event1…			(Repeat counter configuration *LWPMaxEvents* times…)	

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INT	Reserved																									RNH	CNH	DME	BRE	IRE	EN

Figure 10. LWP Flags

Table 9. LWP Flags

Field	Bit	Description
EN	0	LWP is enabled.
IRE	1	Instruction retired event is enabled.
BRE	2	Branch retired event is enabled.
DME	3	DCache miss event is enabled.
CNH	4	CPU clocks not halted event is enabled.
RNH	5	CPU reference clocks not halted event is enabled.
	6:30	Reserved
INT	31	Threshold interrupts are enabled.

2.5 Implementation Notes

The following subsections describe other LWP considerations.

2.5.1 Multiple Simultaneous Events

Multiple events are possible when an instruction retires. For instance, an indirect jump through a pointer in memory can trigger the instructions retired, branches retired, and DCache miss events simultaneously. LWP must count all events that apply to the instruction, but it will only report one event per instruction. The other events will not cause an event record to be stored. The choice of which event to report is implementation dependent and may vary from run to run on the same processor.

This ensures that an instruction that regularly causes multiple events can be reported in all of its categories as the events' counters expire at varying intervals.

2.5.2 Processor State for Context Switch, SVM, and SMM

Implementations of LWP have internal state to hold the current values of the counters for the various events (up to the maximum number of simultaneous events supported), a copy of the pointer into the event buffer, and a copy of the tail pointer for quick detection of threshold and overflow states.

There are times when the system must preserve the volatile LWP state. When the operating system context switches from one user thread to another, the old user state must be saved with the thread's context and the new state must be loaded. When a hypervisor decides to switch from one guest OS to another, the same must be done for the guest systems' states. Finally, state must be stored and reloaded when the system enters and exits SMM, since the SMM code may decide to shut off power to the core.

Hardware cannot maintain the LWP state in the active LWPCB. First, the counters change with every event (not just every reported event), and keeping them in memory would generate a large amount of unnecessary memory traffic. Second, the LWPCB is in user memory and may be paged out to disk at any time, so the memory may not be available when needed.

Fortunately, only the following values need to be preserved when changing threads:

- LWPMSR1—LWPCB address (8 bytes)
- BufferHeadOffset value (4 bytes)
- 26-bit counter values (4 bytes each)
- A flag indicating that the MissedEvents counter must be incremented (1 bit)

In addition, the following values need to be preserved when a hypervisor changes guests or when the system enters low power state:

- LWPMSR0—LWP feature enable flags (4 bytes but can be packed to 1 byte in version 1)
- LWPMSR2—LWP CoreId value (1 byte)

All of the remaining LWP state can be reconstructed from the LWPCB itself.

2.5.2.1 Saving State at Thread Context Switches

If the system includes a hardware feature to extend a system's ability to switch internal system state, that feature will preserve LWP state across context switches. Such a feature requires operating system support for the general mechanism, but once that is done, LWP requires no additional special support.

If implemented, the feature restores the LWP volatile state immediately when restoring other system state. If executed at ring 3, the remaining state gets restored from the LWPCB. If executed when not running at ring 3, the hardware sets a flag and restores the remaining state from the LWPCB at the next transition to ring 3.

If such a hardware feature is not available, LWP internal state can be saved and restored using RDMSR and WRMSR instructions. In this case, this specification will be extended to expose the additional LWP state as MSR addresses. In addition, an LWP driver will require a hook into the OS context switch code.

2.5.2.2 Saving State at SVM Worldswitch to a Different Guest

The VMCB will be augmented to include the volatile LWP state, and VMSAVE and VMLOAD will save and restore that state. The state need not be saved and restored when the guest OS does not change.

2.5.2.3 Saving State at SMM Entry and Exit

The SMM save area will be augmented to include the volatile LWP state. SMM entry and exit will save and restore LWP state as necessary. State must be saved when the processor is going to change power state, but since LWP is ring 3 only, its state should not need to be saved and restored otherwise.

2.5.2.4 Notes on Restoring LWP State

As was mentioned at the top of this section, the LWPCB may not be in memory at all times. Therefore, the LWP hardware must not attempt to access it while still in the OS kernel/VMM/SMM, since that access might fault. The LWP state restore must only be done once the processor is in ring 3 and can take a #PF exception without crashing.

2.5.3 LWPCB Access

Several of the LWPCB fields are written asynchronously by the LWP hardware and by the user software. This section discusses techniques for reducing the associated memory traffic. This is interesting to software because it influences what state is kept internally in LWP, and it helps understand the protocol between the hardware filling the event buffer and the software that will be emptying it.

The hardware can keep internal copies of the buffer head and tail pointers. It need not flush the head pointer to the LWPCB every time it stores an event record; the flush can be deferred until a threshold or buffer full event happens or until context needs to be saved for a context switch. In fact, exceeding the buffer threshold should force the head pointer to memory so that a process polling the ring buffer will be able to see forward progress.

The hardware need not read the software-maintained tail pointer unless it detects a threshold or buffer full condition. At that point, it must reread the tail pointer to see if software has emptied some records from the buffer. If so, it recomputes the threshold condition and acts accordingly. This implies that software polling the buffer should begin processing event records when it detects a threshold event itself. To avoid a race condition with software, the hardware should reread the tail pointer every time it stores an event record while the threshold condition appears to be true. (An implementation can relax this to “every n^{th} time” for some small value of n.) It should also reread it whenever the buffer appears to be full.

The interval values used to reset the counters can be cached in the hardware when the LLWPCB instruction is executed, or they can be read from the LWPCB each time the counter overflows.

The buffer base and buffer size will most likely be cached in the hardware.

The MissedEvents value is intended to be a counter for an exceptional condition, and may be left in memory.

Most cached state can be refreshed from the LWPCB when LWP is enabled either explicitly via LLWPCB or implicitly by having the LWPCB pointer loaded when LWP state is restored.

Caching means that software cannot reliably change sampling intervals or other cached state by modifying the LWPCB. The change might not be noticed by the LWP hardware. On the other hand, changing state in the LWPCB while LWP is running may change the operation at an unpredictable moment in the future if LWP context is saved and restored due to context switching. In summary, software must stop and restart LWP to ensure that any changes reliably take effect.

2.5.4 Security

The operating system must ensure that information does not leak from one process to another or from the kernel to a user process. Hence, if it supports LWP at all, the operating system must ensure that the state of the LWP hardware is set appropriately when a context switch occurs. In a system with hardware context switch support, this should happen automatically. Otherwise, the LWPCB pointer for each process must be saved and restored as part of the process context.

2.5.5 Interrupts

The LWP threshold interrupt address is specified by a Local Vector Table (LVT) entry in the local APIC. This must be set by the operating system to point to the LWP interrupt handler similar to the way the performance counters are connected to their LVT entry. The LWP interrupt is not shared with the performance counter interrupt, since the system allows concurrent and independent use of those two mechanisms. The method of hooking a driver to a local APIC LVT entry is operating system dependent.

2.5.6 TLB and Cache Misses During LWP Logging

When LWP needs to save an event record in the ring buffer, it requires access to the memory containing the ring buffer and sometimes the memory containing the LWPCB. Since these are locations in the user memory space, such access will cause a Page Fault (#PF) exception if these pages are not in memory. A particular implementation of LWP has several ways to handle this situation. Some of these mechanisms may result in reexecuting the instruction or discarding the event and reporting the next event of the appropriate type.

Note that this reinforces the notion that LWP is a sampling mechanism. Programs cannot rely on it to precisely capture every nth instance of an event. It captures *approximately* every nth instance.

Appendix A Glossary

APIC

Advanced Programmable Interrupt Controller—An internal device that can be programmed to handle processor interrupts and direct them to an appropriate interrupt handler.

CPL

Current Privilege Level—The privilege level of the processor, where 0 is the most privileged level and is usually used by the kernel or operating system, and 3 is the least privileged level and is usually used by application programs.

CPUID

An instruction in the x86 architecture that allows a program to determine the features that are present on the current processor.

DCache

Data Cache—The structures in the processor that keep a local copy of data being referenced by the running program. Data in the DCache can be accessed very quickly. There are typically multiple levels of DCache that form a cache hierarchy, with higher cache levels taking more time to access. If a program tries to use data that is not in the DCache, there is typically a long delay while the processor fetches the data from memory or a “farther” level of the cache hierarchy.

DTLB

Data Translation Lookaside Buffer—A TLB structure (see TLB) dedicated exclusively to speeding up access to data by the instructions in a program.

Hypervisor

See VMM.

IBS

Instruction Based Sampling—An extension to the AMD64 architecture introduced in the quad-core AMD Opteron™ processor that can provide performance data that include the precise address of the instruction being sampled, along with details of the execution of the instruction.

ICache

Instruction Cache—The structures in the processor that keep a local copy of instructions being executed by the running program. The ICache can be accessed very quickly. When there are multiple levels of cache hierarchy (see DCache), the first level ICache and DCache often share the other cache levels.

ITLB

Instruction Translation Lookaside Buffer—A TLB structure (see TLB) dedicated exclusively to speeding up access to the instructions in a program.

Kernel mode

Refers to the processor when running at CPL 0, the most privileged level of operation.

LWP

Lightweight Profiling—The hardware proposal in this document to allow performance data to be captured by a program in user mode.

OS

Operating System—The software that provides overall control of the processor. Examples are Microsoft® Windows® and Linux®.

Process

An instance of a program running in a computer. It is started when a program is initiated by a user or by another process. If multiple users are using the same application on a single CPU, there is usually one process for each user.

Retired

An instruction in a processor is retired when all of its operations are complete and the results are committed to the state of the processor. In a complex and out-of-order CPU like the x86, many instructions can be happening simultaneously, but they retire in the original program order.

RIP

The 64-bit instruction pointer register that holds the address of the instruction being executed.

SMM

System Management Mode—An operating mode designed for system control activities that are typically transparent to conventional system software. This includes power management and some low level device control.

SVM

Secure Virtual Machine—The extensions to the AMD64 architecture designed to enable enterprise-class server virtualization software. SVM provides hardware resources that allow a single machine to run multiple operating systems efficiently. See also VMM.

Thread

A flow of instructions associated with a process, usually to perform a particular part of the process' work. A process can have multiple simultaneous threads running to accomplish different parts of its job in parallel.

TLB

Translation Lookaside Buffer—A mechanism to speed up the translation of virtual addresses used by a running program to refer to its memory into physical addresses in the actual main memory of the system.

User mode

Refers to the processor when running at CPL 3, the least privileged level of operation.

VMCB

Virtual Machine Control Block—An area of memory used by SVM and the VMM to hold the state of a guest operating system.

VMM

Virtual Machine Monitor—The software that controls the execution of multiple virtual machines and their *guest* operating systems on a single physical *host* machine. The VMM is responsible for running and switching among the guests and for keeping them isolated from one another.