AMD64 Technology

Lightweight Profiling Specification

Publication No.	Revision	Date
43724	3.08	August 2010

Advanced Micro Devices

Contents

Figures

Tables

Revision History

Date	Revision	Description
August 2010	3.08	Correct description of XRSTOR. Added reserved fields for software in LWPCB. Clarify PRD bit in Branches Retired event.
April 2010	3.06	New encoding for LWP instructions. Removed 16-bit operand size variants. BufferSize in XSAVE area shortened to 28 bits. Clarify details of XRSTOR operation. Change “active” to “enabled” in instruction descriptions. Clarified behaviors when the CPL != 3. Specify reset value of LWP_CBADDR MSR as 0.
August 2009	3.04	Clarified CPUID bits in 3.1, “Detecting LWP Presence” on page 23. Clarified LWP detections in 8.7.1, “System initialization” on page 56. Corrected 8.7.2, “Thread support” on page 56 to refer to LWP_CBADDR.
July 2009	3.03	Second public version.
August 2007	3.00	Initial public version.

1 Introduction

Lightweight Profiling (LWP) is an AMD64 extension to allow user mode 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.

LWP allows a program to gather performance data and examine it either by polling or by taking an occasional interrupt. It introduces minimal additional state to the CPU and the process. LWP differs from the existing performance counters and from Instruction Based Sampling (IBS) because it collects large quantities of data before an taking an interrupt. This substantially reduces the overhead of using performance feedback. An application that polls LWP data requires no interrupts at all.

A program can control LWP data collection entirely in user mode. It can start, stop, and reconfigure profiling without calling the kernel.

LWP runs within the context of a thread, so it can be used by multiple processes in a system at the same time without interference. This also means that if one thread is using LWP and another is not, the latter thread incurs no profiling overhead.

LWP is supported in both long mode and legacy mode.

1.1 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 ring buffer is filled beyond a user-specified threshold, the hardware can cause an interrupt which the operating system (OS) uses to signal a process to empty the ring buffer. With proper OS support, the interrupt can even be delivered to a separate process or thread.

LWP only counts instructions that retire in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. LWP does not count hardware events while the processor is in system management mode (SMM) and while entering or leaving SMM.

Once LWP is enabled, each user-mode thread uses the *LLWPCB* and *SLWPCB* instructions to control LWP operation. These instructions refer to a data structure in application memory called the Lightweight Profiling Control Block, or *LWPCB*, to specify the profiling parameters and to interact with the LWP hardware. The LWPCB in turn points to a buffer in memory in which LWP stores event records.

Each thread in a multi-threaded process must configure LWP separately. A thread has its own ring buffer and counters which are context switched with the rest of the thread state. However, a single monitor thread could collect and process LWP data from multiple other threads.

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 application 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 becomes negative, the event should be reported. 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 may continue to count below zero until the record is written to the event ring buffer.

   For most events, such as instructions retired, LWP gathers an event record describing the instruction that caused the counter to become negative. However, it is valid for LWP to gather event record data for the *next* instruction that causes the event, or to take other measures to capture a 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 become negative, an advanced LWP implementation might gather one event record per event and write them sequentially. A basic LWP implementation may choose one of the eligible events. Other events continue counting but wait until the first event record is written. LWP picks the next eligible instructions for the waiting events. This situation should be extremely uncommon if software chooses large event interval values.
   - LWP may discard an event occurrence. For instance, if the LWPCB or the event ring buffer needs to be paged in from disk, LWP might choose not to preserve the pending event 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 continues counting below 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 the event ring buffer in the process' address space and advances the ring buffer head pointer.
   - LWP checks to see if the ring buffer is full, i.e., if advancing the ring buffer head pointer would make it equal to the tail pointer. If the buffer is full, LWP increments the 64-bit counter LWPCB.MissedEvents. It does not advance the head pointer.
   - If more than one event record reaches the Store stage simultaneously, only one need be stored. Though LWP might store all such event records, it may delay storing some event records or it may discard the information and proceed to choose the next eligible instruction for the discarded event type(s). This behavior is implementation dependent.

- 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 Report and Reset steps (below) may complete later.

4. **Report—**If LWP threshold interrupts are enabled and the space used in the event ring buffer exceeds a user-defined threshold, LWP initiates an interrupt. The OS can use this to signal the process to empty the ring buffer. Note that the interrupt may occur significantly later than the event that caused the threshold to be reached.
5. **Reset—**For each event that was stored, the counter is reset to its programmed interval. If requested by the application, LWP applies randomization to the low order bits of the interval. Counting for that event continues. Reset happens if the ring buffer head pointer was advanced or if the missed event counter was incremented. If the event counter went below -1, indicating that additional events occurred between the selected event and the time it was reported, that overrun value reduces the reset value so as to preserve the statistical distribution of events.

   For all events except the LWPVAL instruction, the hardware may impose a minimum on the reset value of an event counter. This prevents the system from spending too much time storing samples rather than making forward progress on the application. Any minimum imposed by the hardware can be detected by examining the EventInterval*n* fields in the LWPCB after enabling LWP.

An application should periodically remove event records from the ring buffer and advance the tail pointer. (If the application does not process the event records quickly enough or often enough, the LWP hardware will detect that the ring buffer is full and will miss events.) There are two ways to process the gathered events: interrupts or polling.

The application can wait until a threshold interrupt occurs to process the event records 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_CFG—LWP Configuration MSR” on page 27.) One usage model is to associate the LWP interrupt with a semaphore or mutex. When the interrupt occurs, the OS or driver signals the associated object. A thread waiting on the object wakes up and empties the ring buffer. Other models are possible, of course.

Alternatively, the application can have a thread that periodically polls the ring buffer. The polling thread need not be part of the process that is using LWP. It can be in a separate process that shares the memory containing the LWP control block and ring buffer.

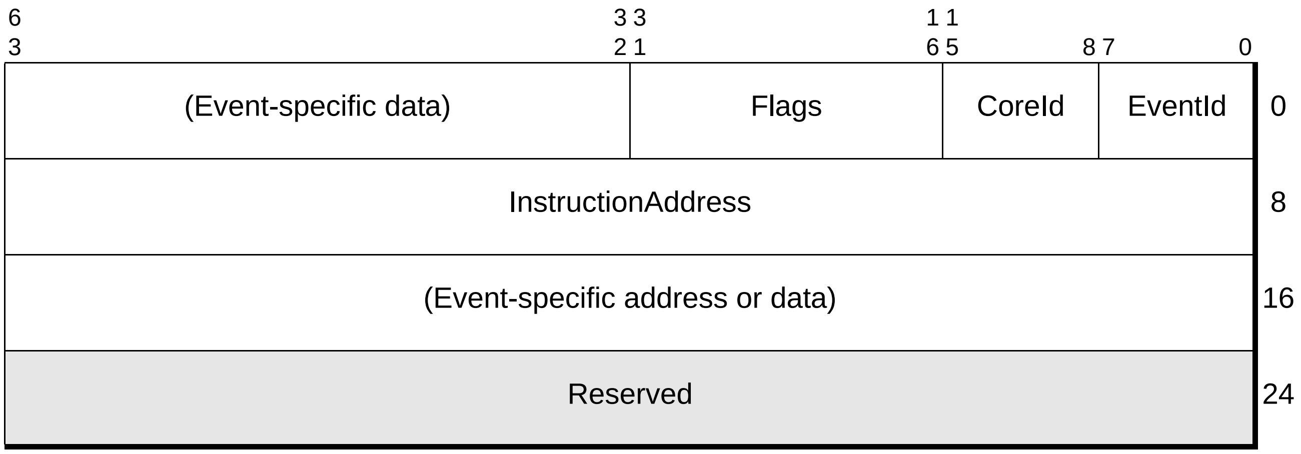
Access to the ring buffer uses a lockless protocol between the LWP hardware and the application. The hardware owns the head pointer and the area in the ring buffer from the head pointer up to (but not including) the tail pointer. (That area might wrap around from the end of the ring to the beginning, of course.) The application must not modify the head pointer nor rely on any data in that region of the ring buffer. If the application has a stale value for the head pointer, it may miss an existing event record but it will never read invalid data. When the application is done emptying the ring buffer, it should refresh its copy of the head pointer to see if the LWP hardware has added any new event records.

Similarly, the application owns the tail pointer and the area in the ring buffer from the tail pointer up to (but not including) the head pointer. The hardware will never modify the tail pointer or overwrite the data in that region of the ring buffer. If the hardware has a stale value for the tail pointer, it may consider that the ring buffer is full or at its threshold, but it will never overwrite valid data. Instead, it refreshes its copy of the tail pointer and rechecks to see if the full or threshold condition still applies.

2 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* (see “Detecting LWP Capabilities” on page 23).

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



Bytes	Field	Description
0	EventId	Event identifier specifying the event record type. Valid identifiers are 1 to 255. 0 is an invalid identifier.
1	CoreId	CPU core identifier value from COREID field of LWP_CFG (see “LWP_CFG—LWP Configuration MSR” on page 27). For multicore systems, this typically identifies the core on which LWP is running. This allows software to aggregate event records from multiple threads into a single data structure without losing CPU information. It also allows software to detect when a thread has migrated from one core to another.
3–2	Flags	Event-specific flags.
7–4		Event-specific data.
15–8	InstructionAddress	The *effective address* of the instruction that triggered this event record. This is the value before adding in the CS base address. If the base is non-zero, software must track it. (Modern operating systems use a CS base of zero, and CS is unused in long mode.)
23–16		Event-specific address or other data.
31–24		Reserved

Figure 2-1. Generic Event Record

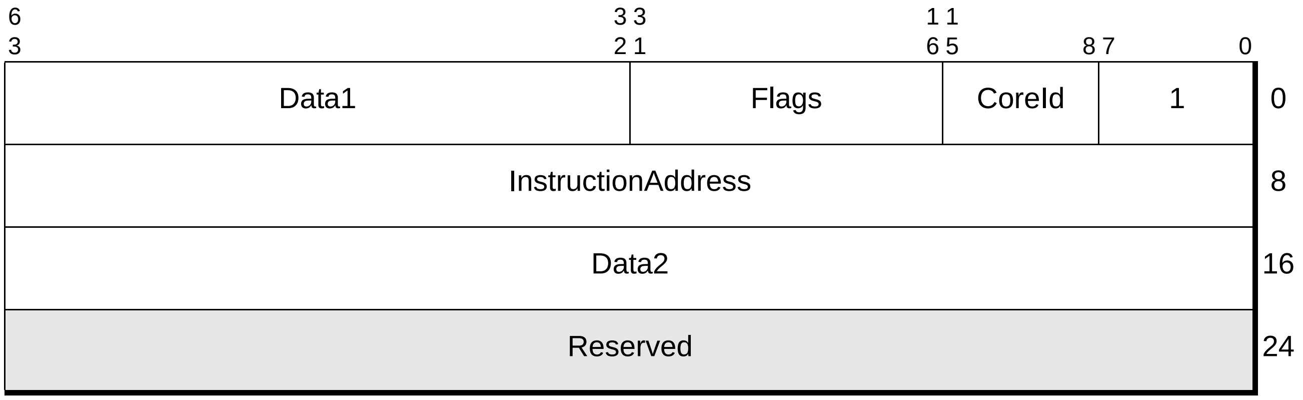
Table 2-1 lists the event identifiers for the events defined in version 1 of LWP. They are described in detail in the following sections.

Table 2-1. EventId Values

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

2.1 Programmed Value Sample

LWP decrements the event counter each time the program executes the LWPVAL instruction (see "LWPVAL—Insert Value Sample in LWP Ring Buffer" on page 32). When the counter becomes negative, it stores an event record with an EventId of 1. The data in the event record come from the operands to the instruction as detailed in the instruction description.



Bytes	Field	Description
0	EventId	Event identifier = 1
1	CoreId	CPU core identifier from LWP_CFG
3–2	Flags	Immediate value (bottom 16 bits)
7–4	Data1	Reg/mem value
15–8	InstructionAddress	Instruction address of LWPVAL instruction
23–16	Data2	Reg value (zero extended if running in legacy mode)
31–24		Reserved

Figure 2-2. Programmed Value Sample Event Record

2.2 Instructions Retired

LWP decrements the event counter each time an instruction retires. When the counter becomes negative, it stores a generic event record with an EventId of 2.

Instructions are counted if they execute entirely in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. All user mode instructions are counted, including LWPVAL and LWPINS.

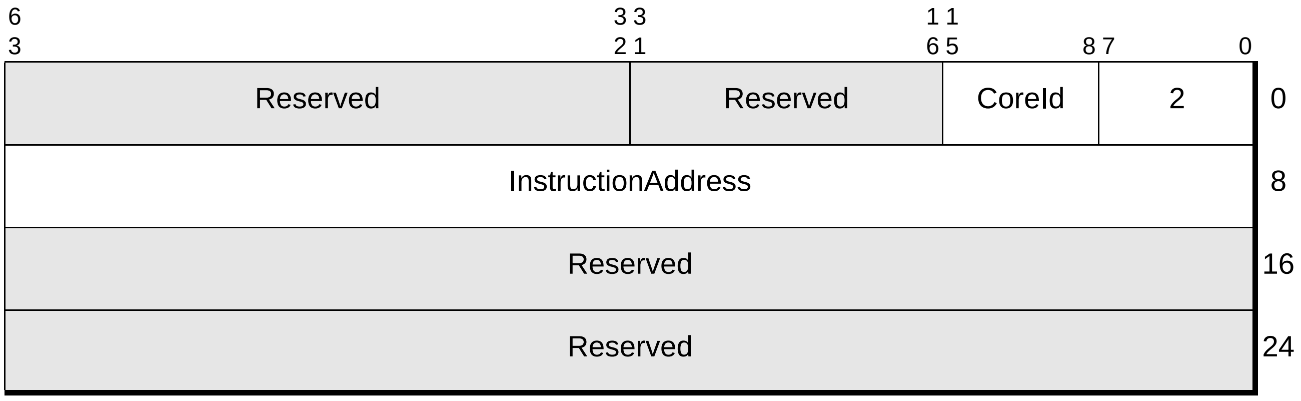


Figure 2-3. Instruction Retired Event Record

2.3 Branches Retired

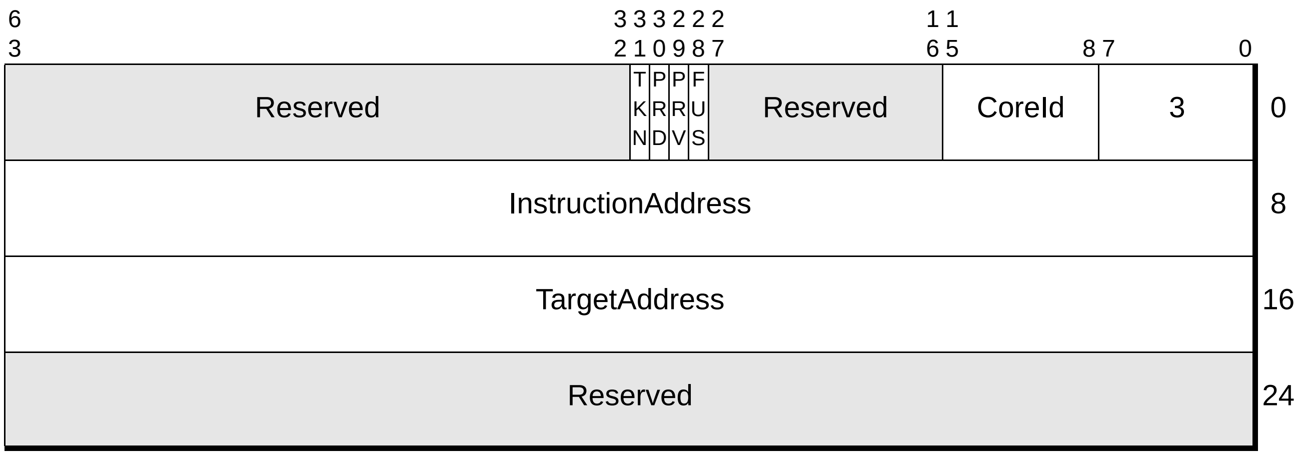
LWP decrements the event counter each time a transfer of control retires, regardless of whether or not it is taken. When the counter becomes negative, it stores an event record with an EventId of 3.

Control transfer instructions that are counted are:

- JMP (near), Jcc, JCXZ, JEXCZ, and JRCXZ
- LOOP, LOOPE, and LOOPNE
- CALL (near) and RET (near)

LWP does not count JMP (far), CALL (far), RET (far), 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, SYSEXIT, SYSRET, VMMCALL, INT, or INTO.

Some implementations of the AMD64 architecture perform an optimization called “fusing” when a compare operation (or other operation that sets the condition codes) is followed immediately by a conditional branch. The processor fuses these into a single operation internally before they are executed. While this is invisible to the programmer, the address of the actual branch is not available for LWP to report when the (fused) instruction retires. In this case, LWP sets the FUS bit in the event record and reports the address of the operation that set the condition codes. If FUS is set, software can find the address of the actual branch by decoding the instruction at the reported InstructionAddress and adding its length to that address. (Note that fused instructions do count as 2 instructions for the Instructions Retired event, since there were 2 x86 instructions originally.)



Bytes	Bits	Field	Description
0		EventId	Event identifier = 3
1		CoreId	CPU core identifier from LWP_CFG
3–2	27–16		Reserved
3	28	FUS	1—Fused operation. InstructionAddress points to a compare operation (or other operation that sets the condition codes) immediately preceding the branch. 0—InstructionAddress points to the branch instruction.
3	29	PRV	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.
3	30	PRD	1—Branch was predicted correctly 0—Mispredicted If PRV = 0, the value of PRD is unpredictable and should be ignored. For unconditional branches, PRD=1 if PRV=1.
3	31	TKN	1—Branch was taken 0—Branch not taken Always 1 for unconditional branches.
7–4			Reserved
15–8		InstructionAddress	Instruction address
23–16		TargetAddress	Address of instruction executed 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. TargetAddress is the *effective address* value before adding in the CS base address.
31–24			Reserved

Figure 2-4. Branch Retired Event Record

2.4 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 come from a level of the cache or memory hierarchy that is selected for counting. When the counter becomes negative, LWP stores an event record with an EventId of 4.

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

2.4.1 Measuring Latency

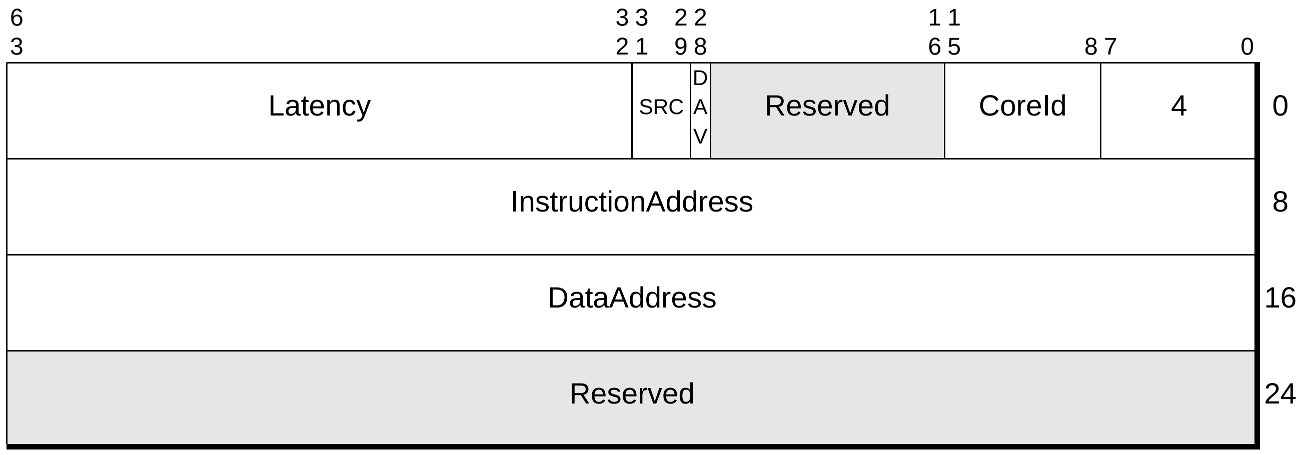
The x86 architecture allows multiple loads to be outstanding simultaneously. An implementation of LWP might not have a full latency counter for every load that is waiting for a cache miss to be resolved. Therefore, an implementation may apply any of the following simplifications. Software using LWP should 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* (see "Detecting LWP Capabilities" on page 23).
- 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 is only 6 bits wide.) The value of *n* is returned as *LwpLatencyMax* (see "Detecting LWP Capabilities" on page 23).

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

2.4.2 Reporting the DCache Miss Data Address

The event record for a DCache miss reports the *linear address* of the data (after adding in the segment base address, if any). 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 becomes negative, or replay the instruction.



Bytes	Bits	Field	Description
0		EventId	Event identifier = 4
1		CoreId	CPU identifier from LWP_CFG
2:3	27–16		Reserved
3	28	DAV	1—DataAddress is valid 0—Address is unavailable
3	31–29	SRC	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)
7–4		Latency	Total latency of cache miss (in cycles)
15–8		InstructionAddress	Instruction address
23–16		DataAddress	Address of memory reference (if flag bit 28 = 1)
31–24			Reserved

Figure 2-5. DCache Miss Event Record

2.5 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 becomes negative, it stores a generic event record with an EventId of 5. This counter varies in real-time frequency as the core clock frequency changes.

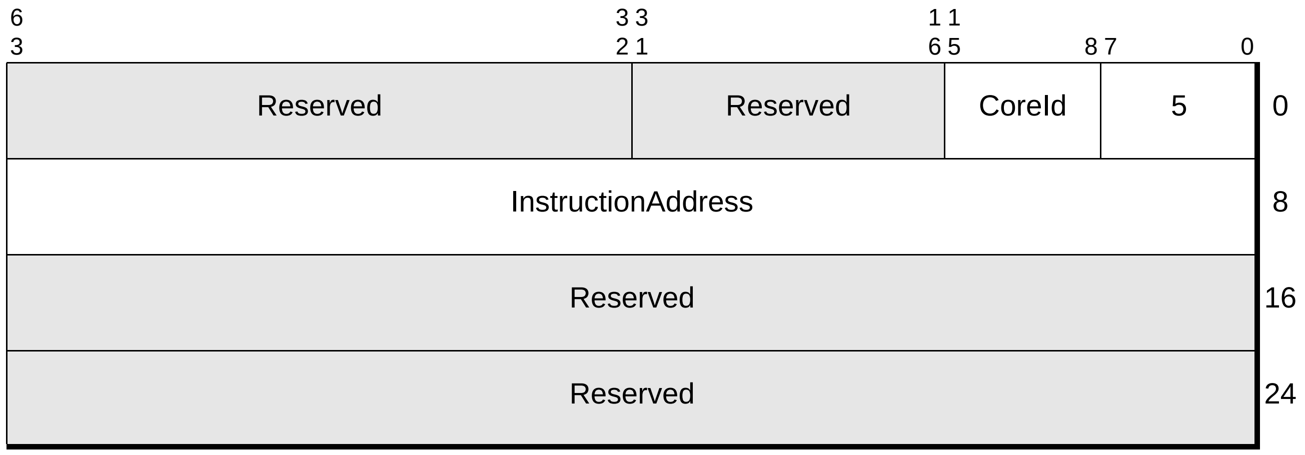


Figure 2-6. CPU Clocks not Halted Event Record

2.6 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 becomes negative, it stores a generic event record with an EventId of 6.

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.

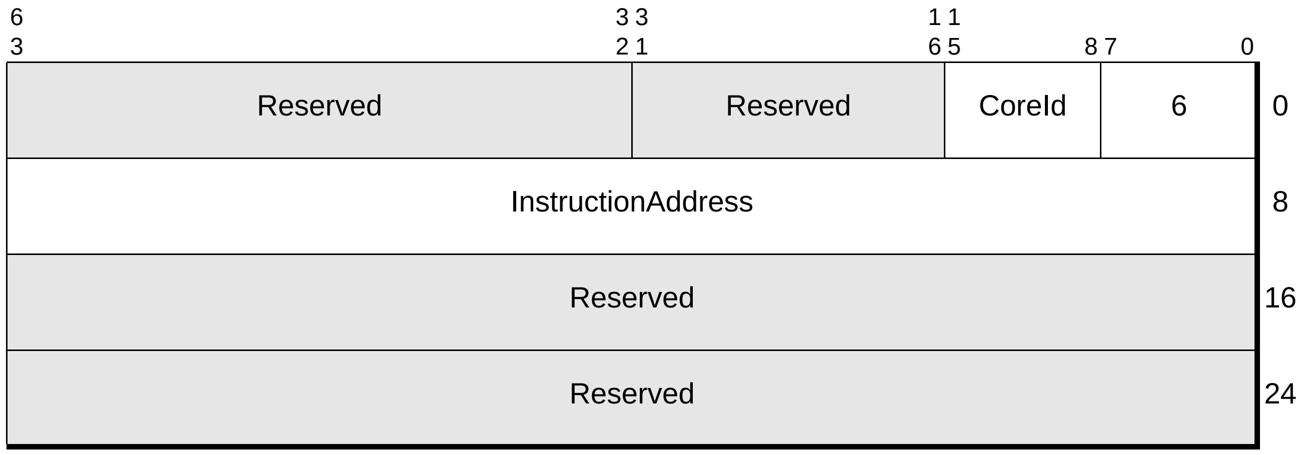
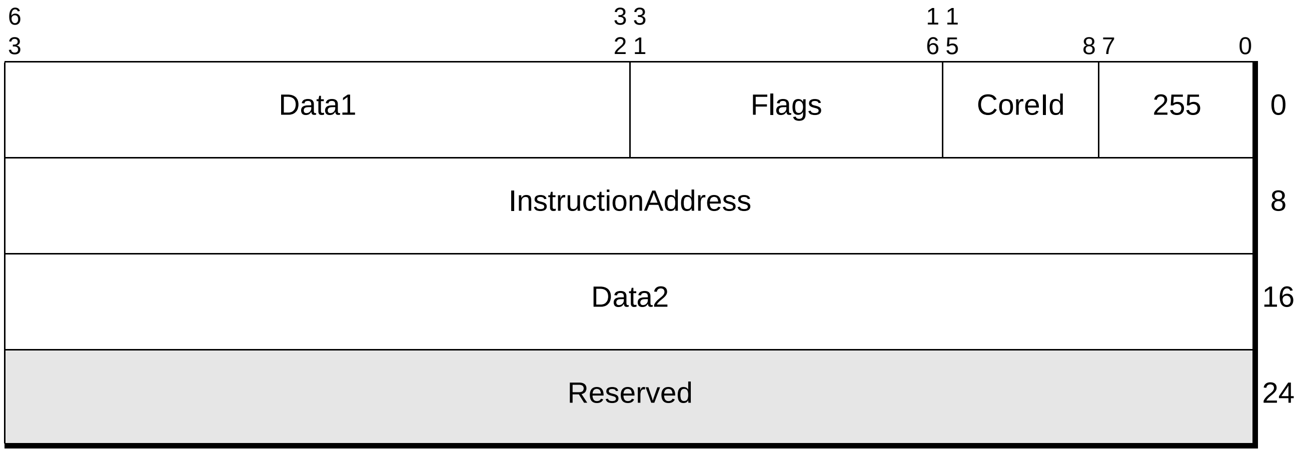


Figure 2-7. CPU Reference Clocks not Halted Event Record

2.7 Programmed Event

When a program executes the LWPINS instruction (see "LWPINS—Insert User Event Record in LWP Ring Buffer" on page 34), the processor stores an event record with an event identifier of 255. The data in the event record come from the operands to the instruction as detailed in the instruction description.



Bytes	Field	Description
0	EventId	Event identifier = 255
1	CoreId	CPU identifier from LWP_CFG
3–2	Flags	Imm16 value
7–4	Data1	Reg/mem value
15–8	InstructionAddress	Instruction address of LWPINS instruction
23–16	Data2	Reg value (zero extended if running in legacy mode)
31–24		Reserved

Figure 2-8. Programmed Event Record

2.8 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

3 Detecting LWP

An application uses the CPUID instruction to identify whether Lightweight Profiling is present and which of its capabilities are *available* for use. An operating system uses CPUID to determine whether LWP is supported on the hardware and to determine which features of LWP are *supported* and can be made *available* to applications.

The notation "CPUID Fn*XXXX_XXXX_RRR*[*FieldName*]" means that the program should execute CPUID with the function code *XXXX_XXXX*h in EAX and then examine the field *FieldName* in register *RRR*. If the "_*RRR*" notation is followed by "_x*YYY*", register ECX must be set to the value *YYY*h before executing CPUID. When *FieldName* is not given, the entire contents of register *RRR* contains the desired value. Numeric values in hexadecimal have an "h" suffix.

3.1 Detecting LWP Presence

LWP is supported on a processor if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. This bit is identical to the value of CPUID Fn0000_000D_EDX_x0[bit 30], which is bit 62 of the XFeatureSupportedMask and indicates XSAVE support for LWP. A system can check either of those bits to determine if LWP is supported. Since LWP requires XSAVE, software can assume that this bit being set implies that CPUID Fn0000_0001_ECX[XSAVE] (bit 26) is also set.

3.2 Detecting LWP XSAVE Area

The size of the LWP extended state save area used by XSAVE/XRSTOR is 128 bytes (080h). This value is returned by CPUID Fn0000_000D_ EAX_x3E (ECX=62).

The offset of the LWP save area from the beginning of the XSAVE/XRSTOR area is 832 bytes (340h). This value is returned by CPUID Fn0000_000D_ EBX_x3E (ECX=62).

The size of the LWP save area is included in the XFeatureSupportedSizeMax value returned by CPUID Fn0000_000D_ECX_x0 (ECX=0).

If LWP is enabled in the XFEATURE_ENABLED_MASK, the size of the LWP save area is included in the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0).

3.3 Detecting LWP Capabilities

The values returned by CPUID Fn8000_001C indicate the capabilities of LWP. See Table 3-1, "Lightweight Profiling CPUID Values" for a listing of the returned values.

Bit 0 of EAX is a copy of bit 62 from XFEATURE_ENABLED_MASK and indicates whether LWP is available for use by applications. If it is 1, the processor supports LWP and the operating system has enabled LWP for applications.

Bits[31:1] returned in EAX are taken from the *LWP_CFG* MSR and reflect the LWP features that are available for use. These are a subset of the bits returned in EDX, which reflect the full capabilities of LWP on current processor. The operating system can make a subset of LWP available if it cannot handle all supported 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 supported capabilities of LWP and make all or some of those features available.

Under SVM, if a VMM allows the migration of guests among processors that all support LWP, it must arrange for CPUID to report the logical AND of the supported feature bits over all processors in the migration pool. Other CPUID values must also be reported as the "least common denominator" among the processors.

Table 3-1. Lightweight Profiling CPUID Values

Reg	Bits	Field	Description
EAX	0	LwpAvail	1—LWP is available to application programs. The hardware and the operating system support LWP. 0—LWP is not available. This bit is a copy of bit 62 of the XFEATURE_ENABLED_MASK register (XCR0).
	1	LwpVAL	LWPVAL instruction (EventId = 1) is available.
	2	LwpIRE	Instructions retired event (EventId = 2) is available.
	3	LwpBRE	Branch retired event (EventId = 3) is available.
	4	LwpDME	DCache miss event (EventId = 4) is available.
	5	LwpCNH	CPU clocks not halted event (EventId = 5) is available.
	6	LwpRNH	CPU reference clocks not halted event (EventId = 6) is available.
	30–7		Reserved for future events.
	31	LwpInt	Interrupt on threshold overflow is available.
EBX	7–0	LwpCbSize	Size in quadwords of the LWPCB. This value is at least (LwpEventOffset / 8) + LwpMaxEvents but an implementation may require a larger control block.
	15–8	LwpEventSize	Size in bytes of an event record in the LWP event ring buffer. (32 for LWP Version 1.)
	23–16	LwpMaxEvents	Maximum supported EventId value (not including EventId 255 used by the LWPINS instruction). Not all events between 1 and LwpMaxEvents are necessarily supported.
	31–24	LwpEventOffset	Offset from the start of the LWPCB to the EventInterval1 field. Software uses 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. LwpEventOffset is always a multiple of 8.

Table 3-1. Lightweight Profiling CPUID Values (Continued)

Reg	Bits	Field	Description
ECX	4–0	LwpLatencyMax	Number of bits in cache latency counters (10 to 31). 0 if DCache miss event is not supported (EDX.LwpDME = 0).
	5	LwpDataAddress	1—Cache miss event records report the data address of the reference. 0—Data address is not reported. 0 if DCache miss event is not supported (EDX.LwpDME = 0).
	8–6	LwpLatencyRnd	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. 0 if DCache miss event is not supported (EDX.LwpDME = 0).
	15–9	LwpVersion	Version of LWP implementation. (1 for LWP Version 1.)
	23–16	LwpMinBufferSize	Minimum size of the LWP event ring buffer, in units of 32 event records. At least 32*LwpMinBufferSize records must be allocated for the LWP event ring buffer, and hence the size of the ring buffer must be at least 32 * LwpMinBufferSize * LwpEventSize bytes. If 0, there is no minimum.
	27–24		Reserved
	28	LwpBranchPrediction	1—Branches Retired events can be filtered based on whether the branch was predicted properly. The values of NMB and NPB in the LWPCB enable filtering based on prediction. 0—NMB and NPB fields of the LWPCB are ignored. 0 if Branches Retired event is not supported (EDX.LwpBRE = 0).
	29	LwpIpFiltering	1—IP filtering is supported. 0—IP filtering is not supported. The IPI, IPF, BaseIP, and LimitIP fields of the LWPCB are ignored.
	30	LwpCacheLevels	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. 0 if DCache miss event is not supported (EDX.LwpDME = 0).
	31	LwpCacheLatency	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. 0 if DCache miss event is not supported (EDX.LwpDME = 0).

Table 3-1. Lightweight Profiling CPUID Values (Continued)

Reg	Bits	Field	Description
EDX	0	LwpAvail	LWP is supported. If 0, the remainder of the data returned by CPUID should be ignored. This bit is a copy of CPUID Fn8000_0001_ECX[LWP] (bit 15).
	1	LwpVAL	LWPVAL instruction (EventId = 1) is supported.
	2	LwpIRE	Instructions retired event (EventId = 2) is supported.
	3	LwpBRE	Branch retired event (EventId = 3) is supported.
	4	LwpDME	DCache miss event (EventId = 4) is supported.
	5	LwpCNH	CPU clocks not halted event (EventId = 5) is supported.
	6	LwpRNH	CPU reference clocks not halted event (EventId = 6) is supported.
	30–7		Reserved for future events.
	31	LwpInt	Interrupt on threshold overflow is supported.

4 LWP Registers

The XFEATURE_ENABLED_MASK register (extended control register XCR0) and the LWP model-specific registers describe and control the LWP hardware. The MSRs are available if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. LWP can only be used if the system has made support for LWP state management available in XFEATURE_ENABLED_MASK.

4.1 XFEATURE_ENABLED_MASK Support

LWP requires that the processor support the XSAVE/XRSTOR instructions to manage LWP state, along with the XSETBV/XGETBV instructions that manage the enabled state mask. An operating system uses XSETBV to set bit 62 of XFEATURE_ENABLED_MASK to indicate that it supports management of LWP state and allows applications to use LWP. When the system makes LWP *available* by setting bit 62 of XFEATURE_ENABLED_MASK, LWP is initially disabled (LWP_CBADDR is zero).

See “Guidelines for Operating Systems” on page 56 for details on how to implement LWP support in an operating system.

4.2 LWP_CFG—LWP Configuration MSR

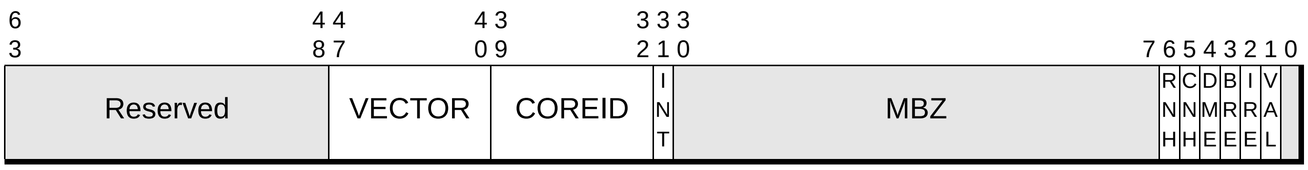
LWP_CFG (MSR C000_0105h) controls which features of LWP are *available* on the processor. The operating system loads LWP_CFG at start-up time (or at the time an LWP driver is loaded) to indicate its level of support for LWP. Only bits for *supported* features (those that are set in CPUID Fn8000_001C_EDX) can be turned on in LWP_CFG. Attempting to set other bits causes a #GP fault.

User code can examine LWP_CFG bits 31:1 by reading CPUID Fn8000_001C_EAX.

Bits 39:32 of LWP_CFG contains the COREID value that LWP will store into the CoreId field of every event record written by this core. The operating system should initialize this value to be the local APIC number, obtained from CPUID Fn0000_0001_EBX[LocalApicId] (bits 31:24). COREID is present so that when LWP is used in a virtualized environment, it has access to the core number without needing to enter the hypervisor. On systems that support x2APIC, local APIC numbers may be more than 8 bits wide. The operating system may then assign LWP COREID values that are small and identify the core within a cluster. If the system has more than 256 cores, there will be unavoidable duplication of COREID values.

Bits 47:40 of LWP_CFG specify the vector number that LWP will use when it signals a ring buffer threshold interrupt.

The reset value of LWP_CFG is 0.



Bits	Field	Description
0		Reserved
1	VAL	Allow the LWPVAL instruction.
2	IRE	Allow LWP to count instructions retired.
3	BRE	Allow LWP to count branches retired.
4	DME	Allow LWP to count DCache misses.
5	CNH	Allow LWP to count CPU clocks not halted.
6	RNH	Allow LWP to count CPU reference clocks not halted.
30–7		Reserved, must be zero
31	INT	Allow LWP to generate an interrupt when threshold is exceeded.
39–32	COREID	Value to store in CoreId field when writing an event record.
47–40	VECTOR	Interrupt vector number to use for LWP Threshold interrupts. Must be provided if INT=1.
63–48		Reserved

Figure 4-1. LWP_CFG—Lightweight Profiling Features MSR

4.3 LWP_CBADDR—LWPCB Address MSR

LWP_CBADDR (MSR C000_0106h) provides access to the internal copy of the *LWPCB linear address*.

RDMSR from this register returns the current LWPCB address without performing any of the operations described for the *SLWPCB* instruction.

WRMSR to this register with a non-zero value generates a #GP fault; use *LLWPCB* or XRSTOR to load an LWPCB address.

Writing a zero to LWP_CBADDR immediately disables LWP, discarding any internal state. For instance, an operating system can write a zero to stop LWP when it terminates a thread.

Note that LWP_CBADDR contains the *linear address* of the control block. All references to the LWPCB that are made by microcode during the normal operation of LWP ignore the DS segment register.

The reset value of LWP_CBADDR is 0. This means that when the system sets bit 62 of XFEATURE_ENABLED_MASK to make LWP *available*, it is initially disabled.

5 LWP Instructions

This section describes the instructions included in the AMD64 architecture to support LWP. These instructions raise #UD if LWP is not *supported* or if bit 62 of XFEATURE_ENABLED_MASK is 0 indicating that LWP is not *available*.

The LLWPCB instruction enables or disables Lightweight Profiling and controls the events being profiled. The SLWPCB instruction queries the current state of Lightweight Profiling.

LWP provides two instructions for inserting user data into the event ring buffer. The LWPINS instruction unconditionally stores an event record into the ring buffer, while the LWPVAL instruction uses an LWP event counter to sample program values at defined intervals.

5.1 LLWPCB—Load LWPCB Address

Parses the Lightweight Profiling Control Block at the address contained in the specified register. If the LWPCB is valid, writes the address into the *LWP_CBADDR* MSR and enables Lightweight Profiling.

Mnemonic	Encoding			
	XOP	RXB.mmmmm	W.vvvv.L.pp	Opcode
LLWPCB *reg32*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	12 /0
LLWPCB *reg64*	8F	$\overline{\text{RXB}}$.09	1.1111.0.00	12 /0

The r/m field of the ModRM byte specifies the register containing the *effective address* of the LWPCB. The mod field of the ModRM byte must be 11b and the vvvv field must be 1111b. The LWPCB address in the register is truncated to 32 bits if the operand size is 32.

The LWPCB must be in memory that is readable and writable in user mode. For better performance, it should be aligned on a 64-byte boundary in memory and placed so that it does not cross a page boundary, though neither of these suggestions is required.

Action

1. If LWP is not *available* or if the machine is not in protected mode, LLWPCB immediately causes a #UD exception.
2. If LWP is already *enabled*, the processor flushes the LWP state to memory in the old LWPCB. See “SLWPCB—Store LWPCB Address” on page 31 for details on saving the active LWP state.

   If the flush causes a #PF exception, LWP remains enabled with the old LWPCB still active. Note that the flush is done before LWP attempts to access the new LWPCB.
3. If the specified LWPCB address is 0, LWP is disabled and the execution of LLWPCB is complete.

4. The LWPCB address is non-zero. LLWPCB validates it as follows:
   - If any part of the LWPCB or the ring buffer is beyond the data segment limit, LLWPCB causes a #GP exception.
   - If the ring buffer size is below the implementation's minimum ring buffer size, LLWPCB causes a #GP exception.
   - While doing these checks, LWP reads and writes the LWPCB, which may cause a #PF exception.

   If any of these exceptions occurs, LLWPCB aborts and LWP is left disabled. Usually, the operating system will handle a #PF exception by making the memory available and returning to retry the LLWPCB instruction. The #GP exceptions indicate application programming errors.
5. LWP converts the LWPCB address and the ring buffer address to *linear address* form by adding the DS base address and stores the addresses internally.
6. LWP examines the LWPCB.Flags field to determine which events should be enabled and whether threshold interrupts should be taken. It clears the bits for any features that are not *available* and stores the result back to LWPCB.Flags to inform the application of the actual LWP state.
7. For each event being enabled, LWP examines the EventInterval*n* value and, if necessary, sets it to an implementation-defined minimum. (The minimum event interval for LWPVAL is zero.) It loads its internal counter for the event from the value in EventCounter*n*. A zero or negative value in EventCounter*n* means that the next event of that type will cause an event record to be stored. To count every j^{th} event, a program should set EventInterval*n* to *j-1* and EventCounter*n* to some starting value (where *j-1* is a good initial count). If the counter value is larger than the interval, the first event record will be stored after a larger number of events than subsequent records.
8. LWP is started. The execution of LLWPCB is complete.

Notes

If none of the bits in the LWPCB.Flags specifies an *available* event, LLWPCB still enables LWP to allow the use of the LWPINS instruction. However, no other event records will be stored.

A program can temporarily disable LWP by executing SLWPCB to obtain the current LWPCB address, saving that value, and then executing LLWPCB with a register containing 0. It can later re-enable LWP by executing LLWPCB with a register containing the saved address.

When LWP is *enabled*, it is typically an error to execute LLWPCB with the address of the active LWPCB. When the hardware flushes the existing LWP state into the LWPCB, it may overwrite fields that the application may have set to new LWP parameter values. The flushed values will then be loaded as LWP is restarted. To reuse an LWPCB, an application should stop LWP by passing a zero to LLWPCB, then prepare the LWPCB with new parameters and execute LLWPCB again to restart LWP.

Internally, LWP keeps the *linear address* of the LWPCB and the ring buffer. If the application changes the value of DS, LWP will continue to collect samples even if the new DS value would no longer allows it to access the LWPCB or the ring buffer. However, a #GP fault will occur if the application uses XRSTOR to restore LWP state saved by XSAVE. Programs should avoid using XSAVE/

XRSTOR on LWP state if DS has changed. This only applies when the CPL != 0; kernel mode operation of XRSTOR is unaffected by changes to DS. See “XSAVE/XRSTOR” on page 47 for details.

Operating system and hypervisor code that runs when the CPL != 3 should use XSAVE and XRSTOR to control LWP rather than using LLWPCB (see below). Use WRMSR to write 0 to LWP_CBADDR to immediately stop LWP without saving its current state (see “LWP_CBADDR—LWPCB Address MSR” on page 28).

It is possible to execute LLWPCB when the CPL != 3 or when SMM is active, but the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Furthermore, if LWP is *enabled* when a kernel executes LLWPCB, both the old and new control blocks and ring buffers must be accessible. Using LLWPCB in these situations is not recommended.

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP is not implemented on this processor.
	X	X		The system is not in protected mode.
			X	LWP is not *available*, or mod != 11b, or vvvv != 1111b.
General protection, #GP			X	Any part of the LWPCB or the event ring buffer is beyond the DS segment limit.
			X	Any restrictions on the contents of the LWPCB are violated
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	LWP was already enabled and a page fault resulted from reading or writing the old LWPCB.
			X	LWP was already enabled and a page fault resulted from flushing an event to the old ring buffer.

5.2 SLWPCB—Store LWPCB Address

Flushes LWP state to memory and returns the current *effective address* of the LWPCB in the specified register.

If LWP is not currently *enabled*, SLWPCB sets the specified register to zero.

The flush operation stores the internal event counters for active events and the current ring buffer head pointer into the LWPCB. If there is an unwritten event record pending, it is written to the event ring buffer.

Mnemonic	Encoding			
	XOP	**RXB.mmmmm**	**W.vvvv.L.pp**	**Opcode**
SLWPCB *reg32*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	12 /1
SLWPCB *reg64*	8F	$\overline{\text{RXB}}$.09	1.1111.0.00	12 /1

The r/m field of the ModRM byte specifies the register in which to put the LWPCB address. The mod field of the ModRM byte must be 11b and the vvvv field must be 1111b. The LWPCB address returned in the register is truncated to 32 bits if the operand size is 32.

If LWP_CBADDR is not zero, the value returned is an effective address that is calculated by subtracting the current DS.Base address from the linear address kept in LWP_CBADDR. Note that if DS has changed between the time LLWPCB was executed and the time SLWPCB is executed, this might result in an address that is not currently accessible by the application.

SLWPCB generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not *available*.

It is possible to execute SLWPCB when the CPL != 3 or when SMM is active, but if the LWPCB pointer is not zero, the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF fault. Using SLWPCB in these situations is not recommended.

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP is not implemented on this processor.
	X	X		The system is not in protected mode.
			X	LWP is not *available*, or mod != 11b, or vvvv != 1111b.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from flushing an event to the ring buffer.

5.3 LWPVAL—Insert Value Sample in LWP Ring Buffer

Decrements the event counter associated with the Programmed Value Sample event (see “Programmed Value Sample” on page 16). If the resulting counter value is negative, inserts an event record into the LWP event ring buffer in memory and advances the ring buffer pointer. If the counter is not negative and the modrm operand specifies a memory location, that location is not accessed.

Mnemonic	Encoding			
	XOP	**RXB.mmmmm**	**W.vvvv.L.pp**	**Opcode**
LWPVAL *reg32.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	0.$\overline{\text{src1}}$.0.00	12 /1 /imm32
LWPVAL *reg64.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	1.$\overline{\text{src1}}$.0.00	12 /1 /imm32

The event record has an EventId of 1. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 2-2 on page 16.

If the ring buffer is not full, the head pointer is advanced and the event counter is reset to the interval for the event (subject to randomization). If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled.

If the ring buffer is full, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, and the head pointer is not advanced.

LWPVAL generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not *available*.

LWPVAL does nothing if LWP is not *enabled* or if the Programmed Value Sample event is not *enabled* in LWPCB.Flags. This allows LWPVAL instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPVAL when the CPL != 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPVAL in these situations is not recommended.

LWPVAL can be used by a program to perform value profiling. This is the technique of sampling the value of some program variable at a predetermined frequency. For example, a managed runtime might use LWPVAL to sample the value of the divisor for a frequently executed divide instruction in order to determine whether to generate specialized code for a common division. It might sample the target location of an indirect branch or call to see if one destination is more frequent than others. Since LWPVAL does not modify any registers or condition codes, it can be inserted harmlessly between any instructions.

Note that when the LWPVAL instruction completes (whether or not it stored an event record in the event ring buffer), it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store an event record. If LWPVAL also stored an event record, the buffer will contain two records with the same instruction address (but different EventId values).

rFLAGS Affected

None

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP is not implemented on this processor.
	X	X		The system is not in protected mode.
			X	LWP is not *available*.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from writing the event to the ring buffer.
			X	A page fault resulted from reading a modrm operand from memory.
General protection, #GP			X	A modrm operand in memory exceeded the segment limit.

5.4 LWPINS—Insert User Event Record in LWP Ring Buffer

Inserts a record into the LWP event ring buffer in memory and advances the ring buffer pointer.

Mnemonic	Encoding			
	XOP	RXB.mmmmm	W.vvvv.L.pp	Opcode
LWPINS *reg32.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	0.$\overline{\text{src1}}$.0.00	12 /0 /imm32
LWPINS *reg64.vvvv, reg/mem32, imm32*	8F	$\overline{\text{RXB}}$.0A	1.$\overline{\text{src1}}$.0.00	12 /0 /imm32

The record has an EventId of 255. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 2-8 on page 22.

If the ring buffer is not full, the head pointer is advanced and the CF flag is cleared. If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled.

If the ring buffer is full, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, the head pointer is not advanced, and the CF flag is set.

LWPINS generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not *available*.

LWPINS simply clears CF if LWP is not *enabled*. This allows LWPINS instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPINS when the CPL != 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPINS in these situations is not recommended.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. For instance, a program might capture 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.

Note that when the LWPINS instruction finishes writing a event record in the event ring buffer, it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store another event record with the same instruction address (but different EventId values).

rFLAGS Affected

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
																M
21	20	19	18	17	16	14	13–12	11	10	9	8	7	6	4	2	0

Note: *Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	LWP is not implemented on this processor.
	X	X		The system is not in protected mode.
			X	LWP is not *available*.
Page fault, #PF			X	A page fault resulted from reading or writing the LWPCB.
			X	A page fault resulted from writing the event to the ring buffer.
			X	A page fault resulted from reading a modrm operand from memory.
General protection, #GP			X	A modrm operand in memory exceeded the segment limit.

6 LWP Control Block

An application uses the LWP Control Block (LWPCB) to specify the details of Lightweight Profiling operation. 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 LWP attempts to access it. To change values in the LWPCB, disable LWP, change the LWPCB (or create a new one), and re-enable LWP.

A few fields are modified by the LWP hardware to communicate progress to the software that is emptying the event ring 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 ring buffer. Software writes these fields and the LWP hardware reads them as needed.

For efficiency, some of the LWPCB fields may be shadowed internally in the LWP hardware unit when profiling is enabled. LWP refreshes these fields from (or flushes them to) memory as needed to allow software to make progress. For more information, refer to “LWPCB Access” on page 54.

The BufferTailOffset field is at offset 64 in the LWPCB in order to place it in a separate cache line on most implementations, assuming that the LWPCB itself is aligned properly. This allows the software thread that is emptying the ring buffer to retain write ownership of that cache line without colliding with the changes made by LWP when writing BufferHeadOffset. In addition, most implementations will use a value of 128 as the offset to the EventInterval1 field, since that places the event information in a separate cache line.

All fields in the LWPCB (as shown in Figure 6-1, “LWPCB—Lightweight Profiling Control Block”) that are marked as “Reserved” (or “Rsvd”) must be zero.

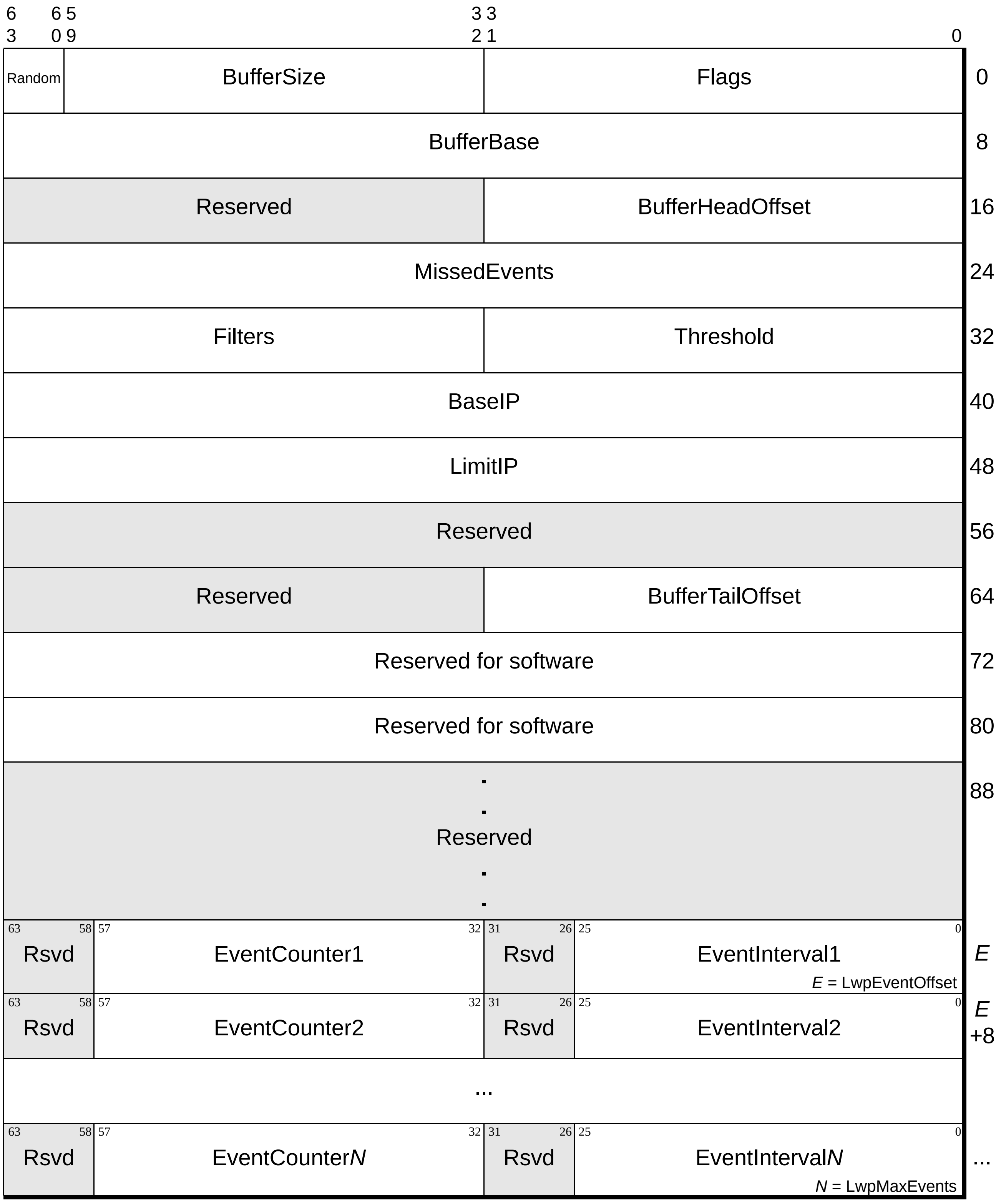


Figure 6-1. LWPCB—Lightweight Profiling Control Block

The R/W column in Table 6-1, “LWPCB—Lightweight Profiling Control Block Fields” indicates how a field is used while LWP is enabled:

- LWP—hardware modifies the field; software may read it, but must not change it
- Init—hardware reads and 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 6-1. LWPCB—Lightweight Profiling Control Block Fields

Bytes	Bits	Field	Description	R/W
3–0		Flags	Flags indicating which events should be or are being counted (see Table 6-2, “LWPCB Flags”) and whether threshold interrupts should be enabled. Before executing *LLWPCB*, the application sets Flags to a bit mask of the events (and interrupt) that should be enabled. LLWPCB does a logical “and” of this mask with the available feature bits in LWP_CFG and rewrites Flags with the mask of features actually enabled.	Init
7–4	59–32	BufferSize	Total size of the event ring buffer (in bytes). Must be a multiple of the event record size *LwpEventSize* (the value used internally will be rounded down if not). BufferSize must be at least (32 * *LwpMinBufferSize* * *LwpEventSize*).	No
7	63–60	Random	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
15–8		BufferBase	The *effective address* of the event ring buffer. Should be aligned on a 64-byte boundary for reasonable performance. Software is encouraged to align the ring buffer to a page boundary for best performance. If the default address size is less than 64 bits, the upper bits of BufferBase must be zero. *LLWPCB* converts BufferBase to a *linear address* and stores it internally. LWPCB.BufferBase is not modified.	No
19–16		BufferHeadOffset	Unsigned offset from BufferBase specifying where the LWP hardware will store the next event record. When BufferHeadOffset == BufferTailOffset, the ring buffer is empty. BufferHeadOffset must always be less than BufferSize; LWP will use a value of 0 if BufferHeadOffset is too large. Also, it must always be a multiple of *LwpEventSize*; LWP will round it down if not.	LWP
23–20			Reserved	

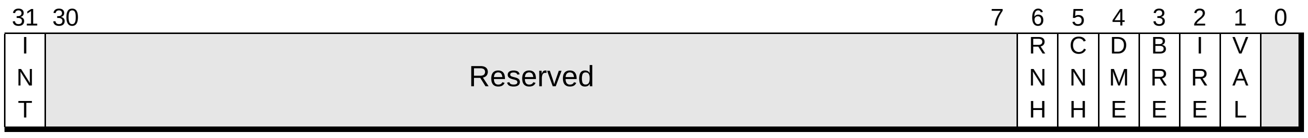
Table 6-1. LWPCB—Lightweight Profiling Control Block Fields (Continued)

Bytes	Bits	Field	Description	R/W
31–24		MissedEvents	The 64-bit count of the number of events that were missed. A missed event occurs when LWP stores an event record, attempts to advance 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 ring buffer is full, the last event record is overwritten.	LWP
35–32		Threshold	Threshold for signaling an interrupt to indicate that the ring buffer is filling up. If threshold interrupts are enabled in *Flags*, then when LWP advances BufferHeadOffset, it computes the space used as ((BufferHeadOffset – BufferTailOffset) % BufferSize). If the space used equals or exceeds Threshold, LWP causes an interrupt. If Threshold is greater than BufferSize, no interrupt will ever be taken. If Threshold is zero, an interrupt will be taken every time an event record is stored in the ring buffer. Threshold is an unsigned integer multiple of *LwpEventSize* (the value used internally will be rounded down if not). Ignored if threshold interrupts are not available in LWP_CFG or if they are not enabled in *Flags*	No
39–36		Filters	Filters to qualify which events are eligible to be counted. This field includes bits to filter branch events by type and prediction status, and bits and values to filter cache events by type and latency. See Figure 6-3, “LWPCB Filters” for details.	
47–40		BaseIP	Low limit of the IP filtering range. An instruction must start at a location greater than or equal to BaseIP to be in range. Ignored if IPF is zero or if the CPUID *LwpIpFiltering* bit is 0 to indicate that IP filtering is not supported.	No
55–48		LimitIP	High limit of the IP filtering range. An instruction must start at a location less than or equal to LimitIP to be in range. Ignored if IPF is zero or if the CPUID *LwpIpFiltering* bit is 0 to indicate that IP filtering is not supported.	No
63–56			Reserved	
67–64		BufferTailOffset	Unsigned offset from BufferBase to the oldest event record in the ring buffer. BufferTailOffset is maintained by software and must always be less than BufferSize and a multiple of *LwpEventSize*. If software stores a value of BufferTailOffset into the LWPCB that violates these rules, the LWP hardware might not detect ring buffer overflow or threshold conditions properly.	SW
71–68			Reserved	
72–87			Reserved for software use. These bytes are never read or written by the LWP hardware	SW

Table 6-1. LWPCB—Lightweight Profiling Control Block Fields (Continued)

Bytes	Bits	Field	Description	R/W
E-1–88			Reserved area between the fixed portion of the LWPCB and the event specifiers. Must be zero. The EventInterval1 field is at offset *E* = *LwpEventOffset*.	
E+3–*E*	25–0	EventInterval1	Reset value for counting events of type EventId = 1 (Programmed Value Sample). A value of *n* specifies that after *n+1* (modified by *Random*) LWPVAL instructions, LWP will store an event record in the ring buffer. EventInterval1 is a signed value. If it is negative, LLWPCB will use zero and will store zero into EventInterval1 in the LWPCB. The Programmed Value Sample event is the only one which allows an interval to be below the implementation minimum interval value.	Init
E+3	31–26		Reserved	
E+7–E+4	57–32	EventCounter1	Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.	LWP
E+7	63–58		Reserved	
E+11–*E+8*	25–0	EventInterval2	Reset value for counting events of type EventId = 2 (Instructions Retired). A value of *n* specifies that after *n+1* (modified by *Random*) instructions are retired, LWP will store an event record in the ring buffer. EventInterval2 is a signed value. If it is negative or is below the implementation minimum, LLWPCB will use the minimum and will store that value into EventInterval2 in the LWPCB.	Init
E+11	31–26		Reserved	
E+15–E+12	57–32	EventCounter2	Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.	LWP
E+15	63–58		Reserved	
		Event3…	Repeat event configuration similar to EventInterval2 and EventCounter2 for EventId values from 3 to *LwpMaxEvents*.	

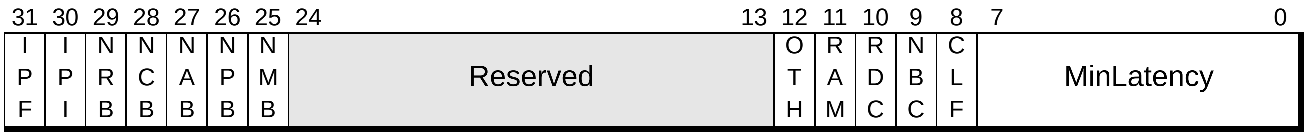
The LLWPCB instruction reads the Flags word from the LWPCB to determine which events to profile and whether threshold interrupts should be enabled. LLWPCB writes the Flags word after turning off bits corresponding to features which are not currently *available*.



Bit	Field	Input to LLWPCB	Value after LLWPCB
0		Reserved	
1	VAL	Enable LWPVAL instruction	LWPVAL instruction enabled
2	IRE	Enable Instructions Retired event	Instructions Retired event enabled
3	BRE	Enable Branches Retired event	Branches Retired event enabled
4	DME	Enable DCache miss event	DCache Miss event enabled
5	CNH	Enable CPU clocks not halted event	CPU Clocks Not Halted event enabled
6	RNH	Enable CPU reference clocks not halted event	CPU Reference Clocks Not Halted event enabled
30–7		Reserved	
31	INT	Enable threshold interrupts.	Threshold interrupts are enabled.

Figure 6-2. LWPCB Flags

Event counting can be filtered by a number of conditions which are specified in the Filters word of the LLWPCB. The IP filtering applies to all events. Cache level filtering applies to all events that interact with the caches. Branch filtering applies to the Branches Retired event.



Bits	Field	Description
7–0	MinLatency	Minimum latency for a cache-related event
8	CLF	Cache level filtering
9	NBC	Northbridge cache events
10	RDC	Remote data cache events
11	RAM	DRAM cache events
12	OTH	Other cache events
24–13		Reserved
25	NMB	No mispredicted branches
26	NPB	No predicted branches
27	NAB	No absolute branches
28	NCB	No conditional branches
29	NRB	No unconditional relative branches
30	IPI	IP filtering invert
31	IPF	IP filtering

Figure 6-3. LWPCB Filters

The following table provides detailed descriptions of the fields in the Filters word.

Table 6-2. LWPCB Filters

Bits	Field	Description
7–0	MinLatency	Minimum latency for a cache-related event to be eligible for LWP counting. Applies to all cache-related events being monitored. MinLatency is multiplied by 16 to get the actual latency in cycles, providing less resolution but a larger range for filtering. An implementation may have a maximum for the latency value. If MinLatency*16 exceeds this maximum value, the maximum is used instead. A value of 0 disables filtering by latency. Ignored if no cache latency event is enabled or if the CPUID *LwpCacheLatency* bit is 0 to indicate that the implementation does not filter by latency (use the CLF bits to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.
8	CLF	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. Cache-related events are only eligible for 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. Ignored if no cache latency event is enabled or if the CPUID *LwpCacheLevels* bit is 0 to indicate that the implementation does not filter by cache level (use the MinLatency field to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.
9	NBC	Northbridge cache events. 1—Count cache-related events that are satisfied from data held in a cache that resides on the Northbridge. 0—Ignore Northbridge cache events Ignored if CLF is 0.
10	RDC	Remote data cache events. 1—Count cache-related events that are satisfied from data held in a remote data cache. 0—Ignore remote cache events. Ignored if CLF is 0.
11	RAM	DRAM cache events. 1—Count cache-related events that are satisfied from DRAM. 0—Ignore DRAM cache events. Ignored if CLF is 0.
12	OTH	Other cache events. 1—Count cache-related events that are satisfied from other sources, such as MMIO, Config space, PCI space, or APIC. 0—Ignore such cache events Ignored if CLF is 0.
24–13		Reserved

Table 6-2. LWPCB Filters

Bits	Field	Description
25	NMB	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. Ignored if the *Branches Retired* event is not enabled or if the CPUID *LwpBranchPrediction* bit is 0 to indicate that the implementation does not filter by prediction.
26	NPB	No predicted branches. 1—Correctly predicted branches will not be counted. Note that since direct branches are always predicted correctly, this is a superset of the NDB filter. 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. Ignored if the *Branches Retired* event is not enabled or if the CPUID *LwpBranchPrediction* bit is 0 to indicate that the implementation does not filter by prediction.
27	NAB	No absolute branches. 1—Absolute branches will not be counted. This only applies to jumps through a register or memory (JMP opcode FF /4) and calls through a register or memory (CALL opcode FF /2). Relative branches (both conditional and unconditional) are counted normally if not disabled via the NRB or NCB bits. 0—Absolute branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the *Branches Retired* event is not enabled.
28	NCB	No conditional branches. 1—Conditional branches will not be counted. This only applies to conditional jumps (Jcc) and loops (LOOPcc). Unconditional relative branches, indirect jumps through a register or memory, and returns are counted normally if not disabled via the NRB or NAB bits. 0—Conditional branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the *Branches Retired* event is not enabled.

Table 6-2. LWPCB Filters

Bits	Field	Description
29	NRB	No unconditional relative branches. 1—Unconditional relative branches will not be counted. This applies to unconditional jumps (JMP), calls (CALL), and returns (RET). Conditional branches and indirect jumps or calls through a register or memory are counted normally if not disabled via the NCB or NAB bits. 0—Direct branches will be counted if not suppressed by other filter conditions. Caution: If NRB, NCB, and NAB are all set, no branches will be counted. Ignored if the *Branches Retired* event is not enabled.
30	IPI	IP filtering invert. 1—IP filtering inverted. Only instructions outside the range from BaseIP to LimitIP are eligible for LWP counting. 0—IP filtering normal. Only instructions inside the range from BaseIP to LimitIP are eligible for LWP counting. Ignored if IPF is zero or if the CPUID *LwpIpFiltering* bit is 0 to indicate that IP filtering is not supported.
31	IPF	IP filtering. 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; instructions at every address are eligible for LWP counting. Ignored if the CPUID *LwpIpFiltering* bit is 0 to indicate that IP filtering is not supported.

7 XSAVE/XRSTOR

LWP requires that the processor support the XSAVE/XRSTOR instructions for managing extended processor state components.

7.1 Configuration

The processor uses bit 62 of XFEATURE_ENABLED_MASK (register XCR0) to indicate whether LWP state can be saved and restored, and thus whether LWP is *available* to applications. The LWP XSAVE area length and offset from the beginning of the XSAVE area are available from the CPUID instruction (see “Detecting LWP XSAVE Area” on page 23). In Version 1 of LWP, the LWP XSAVE area is 128 (080h) bytes long and the offset is 832 (340h) bytes.

7.2 XSAVE Area

Figure 7-1, “XSAVE Area for LWP” shows the layout of the XSAVE area for LWP. It is large enough to allow for future expansion of the number of event counters. Details of the fields are in Table 7-1, “XSAVE Area for LWP Fields”.

All fields in the XSAVE area that are marked as “Reserved” (or “Rsvd”) must be zero.

63 … 32	31 … 28	27 … 0	
LWPCBAddress			0
BufferHeadOffset	Flags		8
BufferBase			16
Filters	Rsvd	BufferSize	24
Saved Event Record			32
			40
			48
			56
EventCounter2	EventCounter1		64
EventCounter4	EventCounter3		72
EventCounter6	EventCounter5		80
Reserved for EventCounter8	Reserved for EventCounter7		88
Reserved for EventCounter10	Reserved for EventCounter9		96
Reserved for EventCounter12	Reserved for EventCounter11		104
Reserved for EventCounter14	Reserved for EventCounter13		112
Reserved for EventCounter16	Reserved for EventCounter15		120

Figure 7-1. XSAVE Area for LWP

Table 7-1. XSAVE Area for LWP Fields

Bytes	Bits	Field	Description
7–0		LWPCBAddress	Address of LWPCB. 0 if LWP is disabled, in which case the rest of the save area is ignored. This is a *linear address*.
8	0		Reserved
8	1	Counter1	1—Event with EventId 1 is active. XRSTOR will make the event active and restore its counter from EventCounter1. 0—Event 1 is not active. XRSTOR will make the event inactive.
9–8	6–2	Counter*n*	Similar to Counter1 for other events.
11–10	31–7		Reserved
15–12		BufferHeadOffset	BufferHeadOffset value
23–16		BufferBase	Address of the event ring buffer. This is a *linear address*.
27–24	27–0	BufferSize	Size of the event ring buffer.
27	31–28		Reserved
31–28		Filters	Profiling filters (same as the Filters field in the LWPCB)
63–32		SavedEventRecord	If an event record is pending, the data to write. May be sparse. Zero in the EventId field means no record pending.
67–64		EventCounter1	Counter for event 1 (if the Counter1 bit is set)
87–68		EventCounter*n*	Counters for events 2–6 (if the respective Counter*n* bit is set)
127–88			Reserved for future event counters

7.3 XSAVE operation

If LWP is not currently enabled (i.e., if LWP_CBADDR = 0), no state needs to be stored. XSAVE sets bit 62 in XSAVE.HEADER.XSTATE_BV to 0 so that an attempt to restore state from this save area will use the processor supplied values. See “Processor supplied values” on page 51.

If LWP is enabled, XSAVE stores the various internal LWP values into the XSAVE area with no checking or conversion and sets bit 62 in XSAVE.HEADER.XSTATE_BV to 1.

7.4 XRSTOR operation

If bit 62 in XFEATURE_ENABLED_MASK (XCR0) is 0 or if bit 62 of EDX:EAX (EDX[30]) is 0, XRSTOR does not alter the LWP state.

If the above bits are 1 but bit 62 in XSAVE.HEADER.XSTATE_BV is 0, XRSTOR writes the LWP state using the processor supplied values, disabling LWP. See “Processor supplied values” on page 51.

If all of the above bits are 1, XRSTOR loads LWP state from the XSAVE area as follows:

1. The internal pointers and sizes are loaded.
   - If BufferSize is below the implementation minimum, LWP is disabled and XRSTOR of LWP state terminates.
   - If BufferSize is not a multiple of the event record size, it is rounded down.
   - If BufferHeadOffset is greater than (BufferSize - LwpEventSize), a value of 0 is used instead.
   - If BufferHeadOffset is not a multiple of the event record size, it is rounded down.
2. For each bit that is set in the Flags field that corresponds to an available event (as currently set in the LWP_CFG MSR), the corresponding event is enabled and the event counter is loaded from the EventCounter*n* field. All other events are disabled.
3. If the EventId field in the SavedEventRecord is non-zero, there was a pending event when XSAVE was executed. XRSTOR loads the event record into hardware. LWP will store it into the event ring buffer as soon as possible once the CPL is 3.

   Software should not alter the SavedEventRecord field. An implementation may ignore a saved event record if it was not constructed by XSAVE. Storing an event into SavedEventRecord and then executing XRSTOR is not a reliable way of injecting an event into the ring buffer.

Note that if LWP is already enabled when executing XRSTOR, the old LWP state is overwritten without being saved.

No interrupt is generated by XRSTOR if the restored value of BufferHeadOffset results in a buffer that is filled beyond the threshold. The interrupt will occur the next time an event record is stored.

XRSTOR may not restore all of the state necessary for LWP to operate. The LWP hardware will read additional state from the LWPCB when it stores then next event record.

If the CPL = 0, XRSTOR simply reloads the LWPCB address and the ring buffer address from the XSAVE area. Kernel software is trusted not to alter the area in such a way as to allow access to memory that the application could not otherwise read or write. The linear addresses in the XSAVE area were validated when the application executed LLWPCB.

If the CPL != 0, XRSTOR first validates the LWPCB and ring buffer pointers. This prevents an application from altering the XSAVE area in order to gain access to memory that it could not otherwise read or write (based on the current values in the DS segment register). Note that if a program’s DS value changes after doing a successful LLWPCB, it might be incapable of doing an XSAVE and then an XRSTOR of LWP state. The XRSTOR will fail if the new DS value no longer allows access to the linear addresses corresponding to the LWPCB or the ring buffer. Programs should avoid this behavior.

If XRSTOR is executed when the CPL != 0, the system performs additional checks on the LWPCB and ring buffer addresses according to the pseudo-code below. A “Store-type Segment_check” fails if the limit check fails (address is beyond the segment limit) or if the segment is read-only.

```
bool Check(uint64 addr, uint32 size) { // Utility function
      if (!64bit_Mode)
            addr = truncate32(addr - DS.BASE)
      uint64 top = addr + size - 1;
```

```
            if (! Store-type Segment_check on DS:[addr] || // Check lower bound
                ! Store-type Segment_check on DS:[top])    // and upper bound
                  return false;
            return true;
      }

      if (! Check(XSAVE.LWPCBAddress, sizeof(LWPCB)) ||
          ! Check(XSAVE.BufferAddress, XSAVE.BufferSize))
            Disable LWP
```

If any of the address checks fails, LWP is disabled. No fault is generated. A program that executes XRSTOR when the CPL != 0 and DS has changed can use SLWPCB to check whether LWP is running.

As with all features that use XSAVE and XRSTOR, if bit 62 of XFEATURE_ENABLED_MASK (XCR0) is 0 but bit 62 of XSAVE.HEADER.XSTATE_BV is 1, XRSTOR will cause a #GP(0) exception.

7.5 Processor supplied values

If XRSTOR is executed when bit 62 of XFEATURE_ENABLED_MASK (XCR0) and EDX:EAX are both 1, but the corresponding bit in XSAVE.HEADER.XSTATE_BV is 0, it indicates that there is no LWP state to restore. In this case, LWP_CBADDR is set to 0 and LWP is disabled. Other processor internal state for LWP is set to 0 as necessary to avoid security issues.

8 Implementation Notes

The following subsections describe other LWP considerations.

8.1 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 counts all events that apply to the instruction, but might not store event records for all events whose event counters became negative. It is implementation dependent as to how many event records are stored when multiple event counters simultaneously become negative. If not all events cause event records to be stored, the choice of which event(s) to report is implementation dependent and may vary from run to run on the same processor.

8.2 Processor State for Context Switch, SVM, and SMM

Implementations of LWP have internal state to hold information such as the current values of the counters for the various events, a pointer into the event ring 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 does not maintain the LWP state in the active LWPCB. This is because the counters change with every event (not just every reported event), so keeping them in memory would generate a large amount of unnecessary memory traffic. Also, 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.

8.2.1 Saving State at Thread Context Switches

LWP requires that an operating system use the XSAVE and XRSTOR instructions to save and restore LWP state across context switches.

XRSTOR restores the LWP volatile state when restoring other system state. Some additional LWP state will be restored from the LWPCB when operations in ring 3 require that information.

LWP does not support the "lazy" state save and restore that is possible for floating point and SSE state. It does not interact with the CR0.TS bit. Operating systems that support LWP must always do an XSAVE to preserve the old thread's LWP context and an XRSTOR to set up the new LWP context. The OS can continue to do a lazy switch of the FP and SSE state by ensuring that the corresponding bits in EDX:EAX are clear when it executes the XSAVE and XRSTOR to handle the LWP context.

8.2.2 Saving State at SVM Worldswitch to a Different Guest

Hypervisors that allow guests to use LWP must save and restore LWP state when the guest OS changes. In addition to the usual information in the VMCB, the hypervisor must use XSAVE/XRSTOR to maintain the volatile LWP state and must also save and restore LWP_CFG. When switching between a guest that uses LWP and one that does not, the hypervisor changes the value of XFEATURE_ENABLED_MASK (XCR0), which ensures that LWP is only enabled in the appropriate guest(s).

A hypervisor need not modify the LWP state if the guest OS is not changed.

8.2.3 Enabling SVM Live Migration

Some hypervisors support live migration of a guest virtual machine. Live migration is when a hypervisor preserves the entire state of the guest running on one physical machine, copies that state to another physical machine, and then resumes execution of the guest on the new hardware.

To allow live migration among machines which may have different internal implementations of LWP, the hypervisor must present the common subset of features among all the hosts in the pool of machines that can be used. Furthermore, since the hypervisor may XSAVE LWP state on one machine and XRSTOR it on another machine, the contents of the XSAVE area must be consistent across all implementations.

This means that an implementation of LWP keeps all event counters internally, not in the LWPCB. If implementations were permitted to differ in this detail, a counter might not get properly restored after migrating the guest machine.

8.2.4 Saving State at SMM Entry and Exit

SMM entry and exit must save and restore LWP state when the processor is going to change power state. SMM must use XSAVE/XRSTOR and must also save and restore LWP_CFG. Since LWP is ring 3 only and is inactive in System Management Mode, its state should not need to be saved and restored otherwise.

8.2.5 Notes on Restoring LWP State

The LWPCB may not be in memory at all times. Therefore, the LWP hardware does not attempt to access it while still in the OS kernel/VMM/SMM, since that access might fault. Some LWP state is restored once the processor is in ring 3 and can take a #PF exception without crashing. This usually happens the next time LWP needs to store an event record into the ring buffer.

8.3 LWPCB Access

Several 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 explains the protocol between the hardware filling the event ring buffer and the software emptying it.

The hardware keeps an internal copy of the event ring buffer head pointer. It need not flush the head pointer to the LWPCB every time it stores an event record. The flush can be done periodically or it can be deferred until a threshold or buffer full condition happens or until the application executes LLWPCB or SLWPCB. Exceeding the buffer threshold always forces the head pointer to memory so that the interrupt handler emptying the ring buffer sees the threshold condition.

The hardware may keep an internal copy of the event ring buffer tail pointer. It need not read the software-maintained tail pointer unless it detects a threshold or buffer full condition. At that point, it rereads the tail pointer to see if software has emptied some records from the ring buffer. If so, it recomputes the condition and acts accordingly. This implies that software polling the ring buffer should begin processing event records when it detects a threshold condition itself. To avoid a race condition with software, the hardware rereads 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 also rereads it whenever the ring 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 ring buffer base and size are cached in the hardware.

The MissedEvents value is a counter for an exceptional condition and is kept in memory.

The cached LWP state is refreshed from the LWPCB when LWP is enabled either explicitly via LLWPCB or implicitly when needed in ring 3 after LWP state is restored via XRSTOR.

Caching implies 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. Software must stop and restart LWP to ensure that any changes reliably take effect.

8.4 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 and when a new process or thread is created. LWP state for a new thread can be initialized by executing XRSTOR with bit 62 of XSAVE.HEADER.XSTATE_BV set to 0 and the corresponding bit in EDX:EAX set to 1.

8.5 Interrupts

The LWP threshold interrupt vector number is specified in the LWP_CFG MSR. The operating system must assign a vector for LWP threshold interrupts and fill in the corresponding entry in the interrupt-descriptor table. Note that the LWP interrupt is not shared with the performance counter interrupt, since the system allows concurrent and independent use of those two mechanisms.

8.6 Memory Access During LWP Operation

When LWP needs to save an event record in the event ring buffer, it accesses the user memory containing the ring buffer and sometimes the memory containing the LWPCB. This causes a Page Fault (#PF) exception if those pages are not in memory.

A particular implementation of LWP has several ways to deal with page faults when storing an event record. These may include saving the event record in the XSAVE area and retrying the store later, 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 n^{th} instance of an event. It captures *approximately* every n^{th} instance.

8.7 Guidelines for Operating Systems

To support LWP, an operating system should follow the following guidelines. Most of these operations should be done on each core of a multi-core system.

8.7.1 System initialization

- Use CPUID Fn0000_0000 to ensure that the system is running on an “Authentic AMD” processor, and then check CPUID Fn8000_0001_ECX[LWP] to ensure that the processor supports LWP.

  Alternatively, check CPUID Fn0000_000D_EDX_x0[bit 30] to ensure that the system supports the LWP XSAVE area, indicating that the processor supports LWP.
- Enable XSAVE operations by setting CR4.OSXSAVE.
- Enable LWP by executing XSETBV to set bit 62 of XCR0.
- Assign a unique interrupt vector number for LWP threshold interrupts and load the corresponding entry in the interrupt-descriptor table with the address of the interrupt handler. This handler should use some system-specific method to forward any threshold interrupts to the application.
- Make LWP available by setting LWP_CFG. To enable all supported LWP features, set LWP_CFG[31:0] to the value returned by CPUID Fn8000_001C_EDX. Set LWP_CFG[COREID] to the APIC core number (or some other value unique to the core) and LWP_CFG[VECTOR] to the assigned interrupt vector number.

8.7.2 Thread support

- For each thread, allocate an XSAVE area that is at least as big as the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0). This is good practice for any system that supports XSAVE.
- When creating a new process or thread, execute XRSTOR with bit 62 of EDX:EAX set to 1 and bit 62 of XSAVE.HEADER.XSTATE_BV set to 0. This ensures that LWP is turned off for any new thread. Alternatively, use WRMSR to write 0 into LWP_CBADDR before starting the thread.

- When saving a running thread’s context, execute XSAVE with bit 62 of EDX:EAX set to 1 to save the thread’s LWP state. It takes almost no time or resources if the thread is not using LWP.
- When restoring a thread’s context, execute XRSTOR with bit 62 of EDX:EAX set to 1. This restores the LWP state for the thread or disables LWP if the thread is not using it.
- When a thread exits or aborts, use WRMSR to store 0 into LWP_CBADDR. This ensures that LWP is turned off.

8.8 Summary of LWP State

LWP adds the following visible state to the AMD64 architecture:

- CPUID Fn8000_0001_ECX[LWP] (bit 15) to indicate LWP support.
- CPUID Fn8000_001C to indicate LWP features.
- Two new MSRs: LWP_CFG, LWP_CBADDR,.
- Four new instructions: LLWPCB, SLWPCB, LWPINS, and LWPVAL.
- Bit 62 in XCR0 (XFEATURE_ENABLED_MASK)
- A new XSAVE area for LWP state.
- New fields for LWP state in the SVM and SMM context, whether in the VMCB and SMM save area or elsewhere.

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.

Available

LWP is available on a processor if it is *supported* on the processor and the system has set XCR0[62]. The XCR0 register is also called XFEATURE_ENABLED_MASK. Bit 62 of that register is visible to the application as CPUID Fn8000_001C_EAX[LwpAvail] (bit 0).

A subsettable feature of LWP (such as threshold interrupts or individual events) is available if LWP is available, the feature itself is *supported*, and the feature's configuration bit in LWP_CFG is set. If a feature is available, the corresponding bit in CPUID Fn8000_001C_EAX is set.

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.

Effective Address

An address in memory that represents an offset into a segmented address space. This is the address of a location before the appropriate segment base address has been added to it. If the segment base is 0 (as it is for most memory references in long mode), this is the same as the linear address.

Enabled

LWP is enabled on a processor if it is *available* on the processor and has been successfully started by executing an LLWPCB or XRSTOR instruction that specifies a non-zero LWPCB address.

A subsettable feature of LWP (such as threshold interrupts or individual events) is enabled if LWP is enabled and the feature was successfully turned on by the LLWPCB or XRSTOR. Features enabled by LLWPCB are reported in LWPCB.Flags.

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 when the CPL = 0, the most privileged level of operation.

Linear Address

An address in memory after any segment base address has been added but before being translated to a physical DRAM address. Also called a *virtual address*.

LWP

Lightweight Profiling—The hardware feature described in this document that allows 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.

Supported

LWP is supported if the hardware is capable of executing the LWP features, indicated by CPUID Fn8000_0001_ECX[LWP] (bit 15) being set. A subsettable feature of LWP is supported if the corresponding bit in CPUID Fn8000_001C_EDX is set.

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 when the CPL = 3, the least privileged level of operation.

Virtual Address

See *Linear Address*.

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.