Lightweight Profiling Proposal

Table of Contents

List of Figures

List of Tables

Change History

2007-07-10	Public proposal version based on 1.01
2007-07-20	Added Glossary

This document and the information contained herein are the proprietary information of Advanced Micro Devices, Inc. (“AMD”). This document is not a contractual document, is provided for informational purposes only, and does not obligate AMD to provide any of the services, technology, or programs described herein. This document is being provided without warranties in its current “AS IS” condition. AMD MAKES NO WARRANTY OF ANY KIND WHATSOEVER, WHETHER EXPRESS OR IMPLIED, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. AMD shall have no liability for the information contained herein, and this document shall not be considered legally binding on either AMD or the recipient, except as may be provided in a separate written agreement.

1 Introduction

The lightweight profiling (LWP) proposal extends the AMD64 architecture (in both legacy and long mode) to allow user mode (CPL=3) processes to gather performance data about themselves with very low overhead. The goal is to enable modules such as dynamic optimizers and managed runtime environments to monitor the currently running program with high accuracy and resolution, thereby allowing them to report on performance problems and opportunities and fix them immediately.

The proposed extensions allow 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. It differs from the existing performance counters and IBS in that large quantities of data are collected with no interrupts, considerably reducing the overhead of using performance feedback. In fact, LWP can be used with a polling scheme that requires no data collection interrupts at all. LWP also allows a user mode program to control its data collection without calling a driver. Finally, LWP runs within the context of a thread, so it can be used by multiple processes within a system at the same time.

2 Requirements

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

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

3 Overview

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

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

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

During profiling, the LWP hardware monitors and reports on one or more types of events. There are several 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, a cache miss event might specify that only events with latency greater than a specified minimum are eligible to be counted.

2. Gather: When an event counter reaches zero, the event should be reported, and LWP gathers an event record. Think of this state as filling in an internal copy of an event record, though actual implementation may vary. The event's counter freezes at zero until an event record is written to the event buffer. (Note that freezing at zero is different behavior from the existing PMU counters.)

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

    - An implementation can choose to gather event information on one or many events at any one time. If multiple event counters reach zero, an advanced LWP implementation may gather one event record per event and write them sequentially. A basic LWP implementation may choose one of the eligible events. The other expired events wait until the chosen event record is written and then pick the next eligible instruction for the waiting event. This situation should be extremely uncommon if software chooses its intervals to be large enough.

    - LWP may on occasion discard an event occurrence. For instance, if the event buffer needs to be paged in from disk, there may be no way to preserve the pending event record data. If an event is discarded, LWP gathers an event record for the next instruction to cause the event.

    - Similarly, if LWP needs to replay an instruction to gather a complete event record, the replay may for some reason abort instead of retiring. The event counter remains zero and LWP gathers an event record for the next instruction to cause the event.

3. Store: When a complete event record has been gathered, LWP stores it into a ring buffer in the process' address space and advances the ring buffer pointer.

    - If the ring buffer is full at this time, LWP increments a 64-bit counter of missed events and does not advance the buffer pointer.

    - If more than one event record reaches the Store stage simultaneously, only one need be stored. LWP may delay storing other event records or it may discard the information and return to choose the next eligible instruction for the discarded event type(s).

- The store need not complete synchronously with the instruction retiring. In other words, if LWP buffers the event record contents, the Store stage (and subsequent stages) may complete some number of cycles after the tagged instruction retires. The data about the event and the instruction are precise, but the rest of the LWP process may complete later.

4. Report: If LWP threshold interrupts are enabled and the space used in the ring buffer exceeds a user-defined threshold, LWP initiates an interrupt so the OS can signal the process to empty the buffer. Note that the interrupt may occur significantly later than the event that caused the threshold to be met.
5. Reset: The counter for the event that was stored is reset to its programmed interval (with any randomization applied). Counting for that event starts again. Reset happens if the event record is stored or if the missed event counter was incremented.

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

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

4 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 currently 32 bytes long. (The actual record size is determined by using CPUID to characterize LWP.)

Reserved fields in event records are set to zero when LWP writes an event record.

Figure 1: Generic Event Record

63 32	31 16	15 8	7 0	
Event-specific data	Flags	CoreId	ID	0
Instruction Address				8
Event-specific address or data				16
Reserved				24

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

Table 1: EventId values lists the event identifiers for the three events available in version 1 of LWP. They are described in detail in the following paragraphs.

Table 1: EventId values

EventId	*Description*
0	Reserved – invalid event
1	Instructions retired
2	Branches retired
3	DCache misses

4.1 Instructions Retired

LWP decrements the event counter each time an instruction retires. When the counter reaches zero, it stores an event record with an event identifier of 1. Bytes 8:15 of the record contain the linear rIP of the instruction whose execution caused the event.

Figure 2: Instruction Retired Event Record

<table>
<tr><th>63 32</th><th>31 16</th><th>15 8</th><th>7 0</th><th></th></tr>
<tr><td>Reserved</td><td>Reserved</td><td>CoreId</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Instruction Address</td><td>8</td></tr>
<tr><td colspan="4">Reserved</td><td>16</td></tr>
<tr><td colspan="4">Reserved</td><td>24</td></tr>
</table>

Bytes	Bits	Description
0		Event identifier = 1
1		CPU core number
2:7		Reserved
8:15		Instruction address.
16:31		Reserved

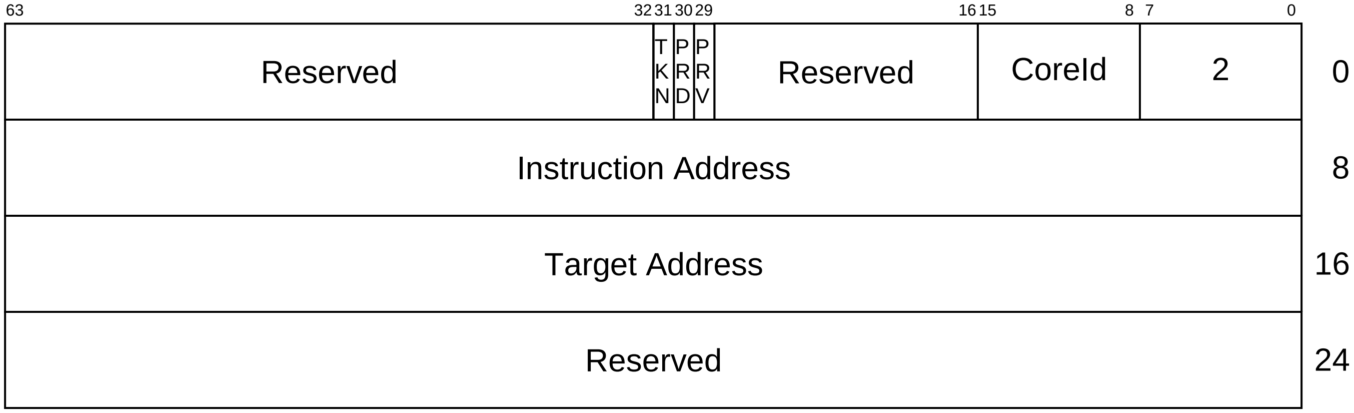
4.2 Branches Retired

LWP decrements the event counter each time a transfer of control retires, regardless of whether or not it is taken. This includes short and long jumps (including JCXZ and its variants), LOOPx, CALL, and RET. It does not include traps or interrupts, whether synchronous or asynchronous, nor does it include operations that switch to or from ring 3 or SMM or SVM, such as SYSCALL, SYSENTER or INT 3.

When the counter reaches zero, LWP stores an event record with an event identifier of 2. The flags indicate whether the branch was taken (always true for unconditional transfers) and whether it was correctly predicted (always true for direct branches). The record also includes the addresses of the branch instruction and the branch target. For not-taken conditional branches, the target is the fall-through address.

Some implementations of LWP might not be able to capture branch prediction information on some or all branches. A bit in the event record indicates whether prediction information is valid.

Figure 3: Branch Retired Event Record



Bytes	*Bits*	*Description*
0		Event identifier = 2
1		CPU core number
2:3	16:29	Reserved
3	29	1 if PRD bit is valid, 0 if prediction information not available
3	30	1 if branch was predicted correctly, 0 if mispredicted
3	31	1 if branch was taken, 0 if not taken
4:7		Reserved
8:15		Instruction address
16:23		Address of instruction after branch. This is the target if the branch was taken and the fall-through address if not.
24:31		Reserved

4.3 DCache misses

LWP decrements the event counter each time a load from memory causes a DCache miss whose latency exceeds the *LWPCacheLatency Threshold* and/or whose data comes from a level of the cache or memory hierarchy that is selected for counting. A misaligned access that causes two misses on a single load only decrements the event counter by 1 and, if it reports an event, the data is for the lowest address that missed. LWP does not count cache misses that are indirectly due to TLB walks, LDT or GDT references, TLB misses, etc. It only counts loads directly caused by the instruction.

When the counter reaches zero, LWP stores an event record with an event identifier of 3. The flags optionally indicate the source of the data, if available. The record includes the total latency, the address of the load instruction, and the address of the referenced data.

Cache misses caused by the LWP hardware itself are not subject to counting.

4.3.1 Measuring latency

Because the architecture allows multiple loads to be outstanding at once, it may be impractical in area and power for an implementation to have a full counter for each load in the MAB that is waiting for a cache miss to be resolved. To make this easier, an implementation may do the following:

- Round the latency to a multiple of 16. In other words, the low 4 bits of reported latency may be 0 and the actual latency counter incremented by 16 every 16 cycles of waiting. (This saves 4 bits per counter at each MAB and only powers the counter about 6% of the time.)
- Approximate the first count. If counting is in increments of 16, the 16 cycles need not start when the load reaches the MAB. Hardware may add 16 to the latency value whenever a convenient internal cycle counter has a carry out of bit 3. (This eliminates the need for a 16 cycle counter per MAB.)
- Cap total latency to 2^n-16 (where n >= 10). The latency counter must be 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. (If n = 10, each counter can be a mere 6 bits wide.) Larger values of n are better, of course, but this is a power/area decision.

  The value of n is returned by the *CPUID instruction* when querying for LWP features.

Also, the latency threshold used to filter events is a multiple of 16 between 0 and 1008, simplifying the comparison that decides whether the cache miss event is eligible.

4.3.2 Reporting the DCache miss data address

The event record for a DCache miss needs to report the linear address of the data.

- If the implementation keeps the linear address with each load that missed in the L1 cache, the complete event record can be created with the load that caused the counter to reach zero.
- If the implementation keeps the linear address, a power saving optimization is possible. The hardware to track the linear address can be kept powered off until the event counter reaches zero. The event that caused the counter to reach zero is not reported. Instead, LWP turns on the address hardware and chooses the next eligible DCache miss to create an event record. Once event counting resumes, the address tracking can be turned off again. If software has decided to get data for every, say, 5000th cache miss, the address tracking gates will only be powered 0.02% of the time. Furthermore, since LWP is a user-mode only feature, the address tracking can be disabled outside ring 3.
- The implementations might not have the linear address available when a load is waiting for data, since physical addresses are all that need be kept around. To deal with this case, LWP does not capture an event record for the instruction that caused the counter to reach zero. Instead, it waits for the next eligible DCache miss to receive its data. At this point, the event counter is already zero, and LWP chooses to report on this new instruction. It saves the RIP and latency of the miss at the time the data returns but before the instruction retires. It then tags the load and aborts it, forcing a pipeline flush. The tagged load gets replayed and, as it proceeds, the tag tells LWP to capture the data linear address, completing the event record. When the load retires, the tag tells LWP to store the event record. The load will not cause a cache miss when replayed, of

course, but since a miss was taken the first time, the event is valid. If for some reason the replay aborts, the event counter remains zero and the next eligible load goes through the same process.

Figure 4: DCache Miss Event Record

63 … 32	31 … 29	28	27 … 16	15 … 8	7 … 0	
Latency	SRC	DAV	Reserved	CoreId	3	0
Instruction Address						8
Data Address						16
Reserved						24

Bytes	*Bits*	*Description*
0		Event identifier = 3
1		CPU core number
2:3	16:27	Reserved
3	28	1 if Data Address is valid, 0 if address is unavailable
3	29:31	Data source for the requested data 0 No valid status 1 Local L3 cache 2 Remote CPU or L3 cache 3 DRAM 4 Reserved (for Remote cache) 5 Reserved 6 Reserved 7 Other (MMIO/Config/PCI/APIC)
4:7		Total latency of cache miss (in cycles) – see text above
8:15		Instruction address
16:23		Address of memory reference (if flag bit 28 = 1)
24:31		Reserved

4.4 Other events

The overall design of LWP allows easy extension to the list of events that it can monitor. The following possibilities come to mind:

- Cycles
- DTLB misses
- FPU operations
- ICache misses
- ITLB misses
- Memory lock contention

5 Details

5.1 CPUID Identification

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

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

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

- Call: CPUID <= EAX: 8000_001C (for extended features, EAX: 8000_001D)
- Return: See table below

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

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

Table 2: Lightweight Profiling CPUID Values

Reg	*Bits*	*Mnemonic*	*Function*
EAX	0	Enabled	LWP is enabled. If this bit is 0, the remainder of the data returned by CPUID should be ignored.
EAX	1	IRE	Instructions retired event (EventId = 1) is enabled
EAX	2	BRE	Branch retired event (EventId = 2) is enabled
EAX	3	DME	DCache miss event (EventId = 3) is enabled

Reg	*Bits*	*Mnemonic*	*Function*
EAX	4:29		Reserved for future events
EAX	30	Extension	Extended CPUID information is available. If 1, information on events with EventId > 29 is available by executing CPUID with EAX = 8000_001D.
EAX	31	Interrupt	Interrupt on threshold overflow is enabled
EBX	7:0	LWPCBSize	Size in bytes of the LWPCB. At least 40 + (LWPMaxEvents * 8) but implementation may require a larger control block
EBX	15:8	LWPEventSize	Size in bytes of an event record in the LWP ring buffer (32 for LWP Version 1).
EBX	23:16	LWPMaxEvents	Number of different events that can be monitored simultaneously.
EBX	31:24	LWPVersion	Version of LWP implementation (1 for LWP Version 1).
ECX	4:0	LWPCacheMax	Number of bits in cache latency counters (10 to 31)
ECX	5	LWPDataAddress	If 1, cache miss event records report the data address of the reference. If 0, data address is not reported.
ECX	29:6		Reserved
ECX	30	LWPCacheLevels	Cache-related events can be filtered by cache level that returned data; the value of CLF in the LWPCB enables cache level filtering. If 0, CLF is ignored. An implementation must support filtering by either latency or cache level. It may support both.
ECX	31	LWPCacheLatency	Cache-related events can be filtered by latency; the value of MinLatency in the LWPCB is used. If 0, MinLatency is ignored. An implementation must support filtering by either latency or cache level. It may support both.
EDX	0	Enabled	LWP is available. If this bit is 0, the remainder of the data returned by CPUID should be ignored.
EDX	1	Interrupt	Interrupt on threshold overflow is available
EDX	2	IRE	Instructions retired event (EventId = 1) is available
EDX	3	BRE	Branch retired event (EventId = 2) is available
EDX	4	DME	DCache miss event (EventId = 3) is available
EDX	5:31		Reserved for future events

5.2 LWPMSRs

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

5.2.1 LWPMSR0 – LWP Feature Enable

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

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

Table 3: LWPMSR0 – Lightweight Profiling Features Enables

Bit	*Mnemonic*	*Function*	*R/W*	*Reset*
0	Enabled	Enable LWP	R/W	0
1	IRE	Allow LWP to count instructions retired	R/W	0
2	BRE	Allow LWP to count branches retired	R/W	0
3	DME	Allow LWP to count DCache misses	R/W	0
4:30		Reserved		0
31	Interrupt	Allow LWP to generate an interrupt when threshold is exceeded	R/W	0

5.2.2 LWPMSR1 – LWPCB

LWPMSR1 provides access to the internal copy of the LWPCB pointer. RDMSR on this register performs the operations described for the SLWPCB instruction, and writing it performs the LLWPCB operations.

5.3 LWP Control Instructions

Use the LLWPCB instruction to enable and disable lightweight profiling and to control the events being profiled. Use the SLWPCB instruction to query the current state of lightweight profiling. These instructions effectively provide user mode access to the LWPCB pointer in LWPMSR1.

LLWPCB –Load LWPCB pointer

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

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

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

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

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

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

Using the previous LWPCB address returned in rAX, a program can temporarily disable LWP by executing LLWPCB with rAX = 0 and then re-enable it by executing LLWPCB with rAX set to the old value.

rFLAGS Affected

None

Exceptions

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

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

SLWPCB – Store LWPCB pointer

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

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

rFLAGS Affected

None

Exceptions

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

5.4 LWP Control Block

The LWP Control Block (LWPCB) specifies the details of how LWP operates. It is an interactive region of memory in which some fields are controlled and modified by the LWP hardware and others are controlled and modified by the software that processes the LWP event records. Of course, hardware will need to cache much of the information from the LWPCB into internal registers for speed.

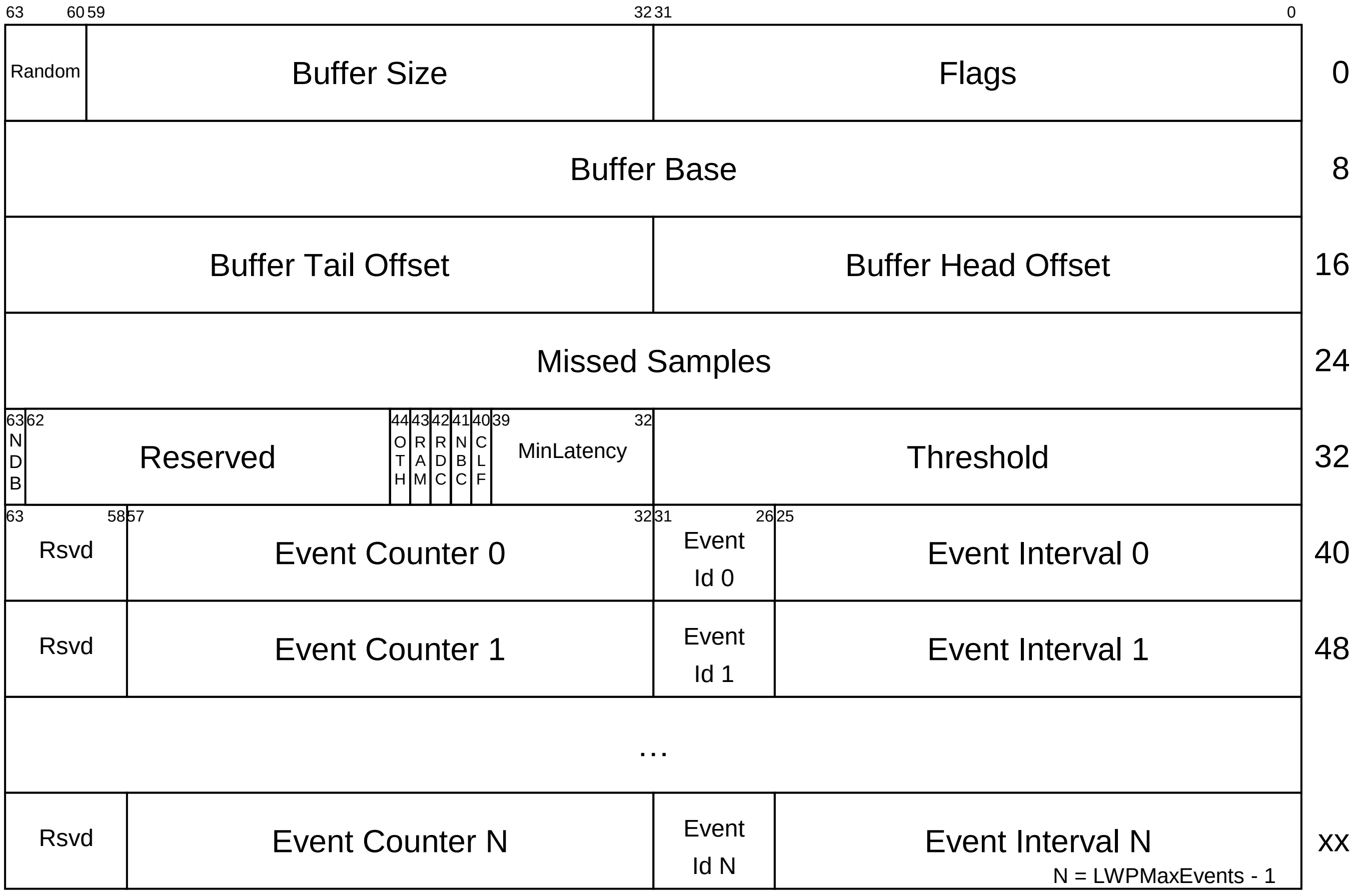
Most of the fields in the LWPCB are constant for the duration of a LWP session (the time between enabling LWP and disabling it). This means that they are loaded into the LWP hardware when it is enabled, and may be periodically reloaded from the same location as needed. The contents of the constant fields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

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

For efficiency, some of the LWPCB fields may be shadowed in registers in the LWP hardware unit when profiling is active. LWP will refresh these fields from (or flush them to) memory as needed to allow software to make progress. See the discussion of *LWPCB Access* later in this document.

The R/W column in *Table 4: LWPCB – Lightweight Profiling Control Block* indicates how the field is modified while LWP is enabled. “LWP” means that hardware modifies the field, “Init” means that hardware modifies the field while executing LLWPCB, “SW” means that software may modify it, and “No” means that the field must remain unchanged as long as the LWPCB is in use.

Figure 5: Lightweight Profiling Control Block (LWPCB)



LWPCB Size = 40 + (LWPMaxEvents * 8)

Table 4: LWPCB – Lightweight Profiling Control Block

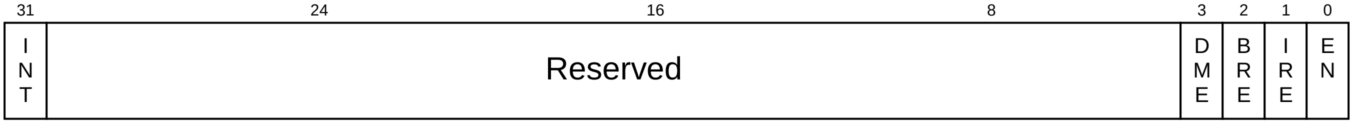
Field	*Description*	*R/W*
Flags	Flags indicating LWP state (see Figure 6: LWP Flags).	Init
BufferSize	Total size of the event record buffer (in bytes). Must be a multiple of the event record size (LWPEventSize).	No
Random	Number of bits of randomness to use in counters. Each time a counter is loaded from an interval to start counting down to the next event record, the bottom Random bits are set to a random value. This avoids fixed patterns in events.	No

Field	Description	R/W
BufferBase	Linear address of the event record buffer. Must be aligned on a 32-byte boundary (the low 5 bits of BufferBase are ignored). Software is encouraged to align the buffer on a page boundary, but this is not required.	No
BufferHeadOffset	Unsigned offset into BufferBase specifying where the LWP hardware will store the next event record. When BufferHeadOffset == BufferTailOffset, the buffer is empty. BufferHeadOffset is always < BufferSize and is always a multiple of LWPEventSize.	LWP
BufferTailOffset	Unsigned offset into BufferBase specifying the oldest event record in the buffer. BufferTailOffset is always < BufferSize and is always a multiple of LWPEventSize.	SW
MissedEvents	64-bit count of the number of events that were missed. A missed event occurs after LWP stores an event record when it increments BufferHeadOffset and discovers that it would be equal to BufferTailOffset. Instead of updating BufferHeadOffset, LWP increments the MissedEvents counter. Thus, when the buffer is full, the last event record is overwritten.	LWP
Threshold	If non-zero, threshold for interrupting the user to indicate that the buffer is filling up. When LWP advances BufferHeadOffset, it computes the space used as ((BufferHeadOffset – BufferTailOffset) % BufferSize) and compares it to Threshold. If the space used == Threshold and threshold interrupts are enabled, LWP causes an interrupt. (Note that no division is needed for the modulus operator; if the difference is < 0, simply add BufferSize to the difference.) The compare for equality ensures that only one interrupt occurs when the threshold is crossed. If zero, no threshold interrupts will be generated. This field is ignored if threshold interrupts are not enabled in LWPMSR1.	No

Field	***Description***	***R/W***
MinLatency	Minimum latency required to make a cache-related event eligible for LWP counting. Applies to all cache-related events being monitored. The number in MinLatency is multiplied by 16 to get the actual latency in cycles. This scaling provides less resolution but a larger range for filtering. An implementation may have a maximum for the latency value it captures. If MinLatency*16 exceeds this maximum value, the maximum is used instead. A value of 0 disables filtering by latency. NOTE • MinLatency is ignored if no cache latency event is chosen in one of the EventId*n* fields. • MinLatency is ignored if CPUID indicates that the implementation does not filter by latency. Use the CLF bits to get a similar effect. At least one of these mechanisms must be available.	No
CLF	Cache Level Filtering. 1 enables filtering cache-related events by the cache level or memory level that returned the data. It enables the next 4 bits, and cache-related events are only eligible for LWP counting if the bit describing the memory level is on. 0 means no cache level filtering, the next 4 bits are ignored, and any cache or memory level is eligible. NOTE • CLF is ignored if no cache latency event is chosen in one of the EventId*n* fields. • CLF is ignored if CPUID indicates that the implementation does not filter by cache level. Use the MinLatency field to get a similar effect. At least one of these mechanisms must be available.	No
NBC	Set to 1 to record cache-related events that are satisfied from data held in a cache that resides on the Northbridge. Ignored if CLF is 0.	No
RDC	Set to 1 to record cache-related events that are satisfied from data held in a remote data cache. Ignored if CLF is 0	No
RAM	Set to 1 to record cache-related events that are satisfied from DRAM. Ignored if CLF is 0	No
OTH	Set to 1 to record cache-related events that are satisfied from other sources, such as MMIO, Config space, PCI space, or APIC. Ignored if CLF is 0	No

Field	Description	R/W
NDB	No direct branches. 1 means direct branches will not be counted. This only applies to unconditional RIP-relative branches. Conditional branches, indirect jumps through a register or memory, calls, and returns are counted normally. This value is ignored if the *Branches Retired* event is not chosen in one of the EventId*n* fields.	No
EventInterval0	Number of events of type EventId0 to count before storing an event record.	No
EventId0	EventId of the event to count in this counter. 0 means disable this counter. It is invalid to specify the same EventId in two or more counters and may cause unpredictable results.	No
EventCounter0	Starting or current value of counter	LWP
Event1…	(Repeat counter configuration LWPMaxEvents times…)	

Figure 6: LWP Flags



Field	Bit	Description
EN	0	LWP is enabled
IRE	1	Instruction retired event is enabled
BRE	2	Branch retired event is enabled
DME	3	DCache miss event is enabled
	1:30	Reserved
INT	31	Threshold interrupts are enabled

5.5 Implementation Notes

5.5.1 Multiple simultaneous events

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

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

5.5.2 Processor State for context switch, SVM, and SMM

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

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

It is not sufficient to simply keep all LWP state in the active LWPCB. First, the counters change with every event (not just every reported event), and keeping them in memory would generate a large amount of unnecessary memory traffic. More problematic is the fact that the LWPCB is in user memory and may be paged out to disk at any time, so the memory may not be available when needed.

Fortunately, the amount of state to be preserved is quite small:

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

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

5.5.2.1 Saving state at thread context switches

The most effective way to preserve LWP state across context switches is to add a hardware feature to extend a system's ability to switch internal system state. This would require kernel support for the general mechanism, but once that is done, LWP requires no additional special support.

Such a mechanism, if implemented, should restore the LWP volatile state immediately. If executed at ring 3, the remaining state gets restored from the LWPCB. If executed when not running at ring 3, the hardware sets a flag and restores the remaining state from the LWPCB at the next transition to ring 3.

Otherwise, the kernel or a driver must take care of saving and restoring state and we will need to add a way to access the LWP internal state from native code. (Unfortunately, in Microsoft Windows® there is no way for a driver to hook into the system context switch.)

5.5.2.2 Saving state at SVM worldswitch to a different guest

Augment the contents of the VMCB to include the volatile LWP state. VMSAVE and VMLOAD will save and restore that state, and could leverage the hardware mechanism described above for context switches. Note that with a well written VMM, this avoids saving and restoring state when the guest OS does not change.

5.5.2.3 Saving state at SMM entry and exit

Augment the contents of the SMM save area as above. SMM entry and exit will save and restore LWP state, using the same ring-3 transition flag as above.

5.5.2.4 Notes on restoring LWP state

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

5.5.3 LWPCB Access

Several of the LWPCB fields are written asynchronously by the LWP hardware and by the user software. This section discusses techniques for reducing the associated memory traffic.

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

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

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

The buffer base and buffer size should be cached, but can be refreshed from the LWPCB when LWP is enabled either explicitly via LLWPCB or implicitly by having the LWPCB pointer loaded when LWP state is restored.

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

5.5.4 Security

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

5.5.5 Interrupts

Microsoft Vista® no longer allows a system to hook any APIC interrupt that it does not already know about. To do so now requires a custom HAL, which is untenable for a system we want to deploy. We need a TBD method of interrupting (as does, in fact, the traditional performance counter interface). So let's figure one out, or create a new one, or share an existing interrupt.

5.5.6 TLB and Cache misses during LWP logging

When LWP decides to save an event record in the ring buffer, it requires access to the memory containing the ring buffer and sometimes the memory containing the LWPCB. Since these are locations in the user memory space, such access will cause a Page Fault (#PF) exception if these pages are not in memory. There are several ways that the LWP hardware can handle this:

- Generate a fault. Abort the current instruction and any update to the LWP internal state and generate a #PF exception for the buffer or LWPCB address. When the #PF exception handler returns, the instruction will be executed again. If the event record was for instructions retired or for branches retired, the event will occur again and the event record can now be stored. If the record is for DCache misses, the particular event that caused the fault will likely not recur, and the next DCache miss will trigger LWP.
- Generate a trap. Retire the current instruction but avoid updating LWP internal state, then generate a #PF exception. The buffer and/or LWPCB will be paged into memory; the next time the event occurs, LWP will most likely be able to complete its memory transactions.
- Generate a trap, but augment the CPU and/or LWP state so that the memory transaction information is kept around. Arrange for the LWP hardware to run "between" the counted instruction and the instruction executed when the trap returns, at which time it re-executes the memory transaction that faulted.
- Require that the buffer and LWPCB be in non-paged memory. This seems particularly onerous for a feature that we expect to be deployed in many threads, since each thread using LWP would lock down a substantial chunk of memory. Possible, but not recommended.
- Lighten the problem a bit by reworking this proposal so that the LWPCB is never referenced after the LLWPCB instruction is issued. This reduces the problem to only the event buffer memory region. The LWP state would need to be context switched and we'd need to figure out a way to tell LWP when software has processed the buffer and advanced the tail pointer. This might be challenging if the routine emptying the buffer runs in a different thread.

In pathological cases, page faults during LWP may thrash and prevent events from ever being stored, but this is unlikely enough to be acceptable.

Note that this approach 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.

Appendix A: Glossary

APIC

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

CPL

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

CPUID

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

DCache

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

DTLB

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

HAL

Hardware Abstraction Layer – A software layer in Microsoft Windows® that is intended to abstract differences between microprocessor implementations away from the operating system.

Hypervisor

See VMM.

IBS

Instruction Based Sampling – An extension to the AMD64 architecture introduced in the Barcelona family of microprocessors that can provide performance data that include the precise address of the instruction being sampled, along with details of the execution of the instruction.

ICache

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

ITLB

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

Kernel mode

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

LWP

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

MAB

Miss Address Buffer – A structure in the AMD64 processor that holds operations that missed in the DCache when attempting to access memory. Operations wait in the MAB until the data is available.

OS

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

Process

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

Retired

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

RIP

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

SMM

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

SVM

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

Thread

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

TLB

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

User mode

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

VMCB

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

VMM

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