Introduction to Microarchitectural Optimization for Itanium® 2 Processors

Reference Manual

Copyright © 2002 Intel Corporation
All Rights Reserved
Issued in U.S.A.
Document Number: 251464-001

World Wide Web: http://developer.intel.com

Version	Version Information	Date
-0	First publication	07/ 2002

This manual as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS. INTEL MAY MAKE CHANGES TO SPECIFICATIONS AND PRODUCT DESCRIPTIONS AT ANY TIME, WITHOUT NOTICE.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel, the Intel logo, Pentium, Itanium, MMX, and Intel XScale are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2002 Intel Corporation.

Contents

Chapter1	Introduction	1
Chapter2	Microarchitecture Tuning Overview	3
Chapter3	Itanium® 2 Processor Architecture	5
Chapter4	Event-based Sampling with the VTune™ Performance Analyzer	15
Chapter5	Performance Monitoring and Cycle Accounting	17
Chapter6	BE_EXE_Bubble for Memory Access Stalls in the EXE Pipeline Stage	31
Chapter7	BE_L1D_FPU_Bubble for Stalls due to L1D and FPU Micropipelines	57
Chapter8	BE_Flush_Bubble for Stalls due to Pipeline Flushes	65
Chapter9	BE_RSE_Bubble for Stalls due to the Register Stack Engine	71
Chapter10	Back_End_Bubble.FE for Stalls due to the Pipeline Front End	75
	Appendix	77
	Glossary	85
	Index	91

Introduction

1

This guide introduces software developers to the systematic use of Itanium® 2 processor performance monitoring events to analyze the execution efficiency of their application using the VTune™ Performance Analyzer. While several Intel reference documents (see references at the end of this chapter) discuss the performance counters, software optimization and the VTune analyzer separately in detail, this guide focuses on a global discussion of microarchitectural optimization using the performance counters, the VTune analyzer and the compilers. This guide is therefore not intended as a replacement of the references but to serve as an introduction to these materials.

The intended readers of this guide are software developers working on performance critical applications, software development tools, device drivers and operating systems.

The Itanium® Processor Family architecture relies on explicit scheduling for achieving the highest performance. Consequently, an essential component of software development for these new processors is developing code in high-level languages and using the most advanced versions of the compilers. The focus of this guide is on optimizing code developed in high-level languages. The optimizations developed for the Itanium® 2 processor will apply well for future generations, with at most a recompilation to incorporate future architectural advances. While the microarchitectural optimization methodology is perfectly applicable to hand-coded assembler, hand-coded assembler should be avoided as it cannot be rescheduled by a compiler for future architectures.

This edition of the guide will discuss overcoming execution inefficiencies caused by improper matching of algorithm and data structures with the microarchitecture encountered during data reading operations. It will not discuss execution inefficiencies encountered during write operations. That will be covered in a future revision.

References

1.Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0

2. Itanium® Architecture for Software Developers by W. Triebel, Intel Press

3. VTune™ Performance Analyzer online help

4.Software Optimization for High Performance Computing by K.R Wadleigh and I. L. Crawford, Hewlett-Packard* Professional Books

5.Itanium® 2 Processor Microarchitectural Specification

6.Intel® Itanium® Architecture Software Developers Manual, Volumes 1 through 3

Microarchitecture Tuning Overview

2

Microarchitectural performance tuning is the exercise of maximizing the execution efficiency which means the flow of instructions through the processor's functional units. As such, some of the implicit assumptions are as follows:

- the algorithm is sensibly coded
- an optimal trade-off of space for speed has been arrived at
- there is sufficient memory so that the application is not making excessive disk accesses
- peripheral devices that are required (network interfaces, video, etc.) have sufficient bandwidth to not be the source of performance limitations.

Further, the analysis must be performed on a machine that has been correctly configured.

Using the performance monitoring counters and a performance analysis tool to collect the counter data, you can find out what is restricting the flow of instructions to the functional units. The VTune™ Performance Analyzer is probably the most widely used tool of this type and provides a rich spectrum of uses. However, the VTune analyzer collects counter data using exception handlers and this is quite intrusive to the normal processor execution. For many analyses, global information is sufficient so you can use less intrusive tools like Emon to gather data.

In all cases, the performance analysis tools sample the entire machine, so dedicated use is required to collect meaningful data. If there are multiple users or uses of the analysis machine while the data collection is in progress, there is a risk of data collection distortion due to the OS multi-tasking between the processes.

In order to correctly interpret the performance counter data, you must have a good understanding of the processor's microarchitecture. The performance counters record the use and invocation of the processor's architectural features during the application execution. As such, they help determine the sources of inefficiencies in the application's interaction with the microarchitecture. You must be familiar with a wide spectrum of performance counters and the precise features of the architecture they measure in order to use them efficiently to analyze the execution inefficiencies of your application.

This Software Optimization guide is organized to present the microarchitecture, the performance analysis tools, and the systematic use of performance counters. More extensive documentation exists on each of these topics individually in other documents (see the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0). However, this guide will focus on the following:

- Microarchitectural aspects that impact an application's execution efficiency. In particular, performance counters that help you monitor the processor activity and identify microarchitectural performance bottlenecks.
- The VTune Performance analysis tool and its ability to collect performance monitor data during the execution of the application.

The systematic use of the performance monitors organized around the Itanium® processor family's cycle accounting capability. On the Itanium 2 processors, cycle accounting events can be used to define a methodology for determining the dominant execution inefficiencies. The bulk of this guide is an exposition of this analysis methodology and the typical prescriptions for removing the execution inefficiencies through source code and or compilation modification.

Itanium® 2 Processor Architecture

3

In order to optimize code to take advantage of microprocessor architectural features, you must-understand the target architecture. The subject of this treatment is the efficient consumption and execution of compiled high-level language code by the microprocessor. You, as a software developer are not a passive player in this process but can actively reorganize data and algorithms to take advantage of architectural capabilities and avoid pitfalls due to simple oversights.

The following discussion is not intended to be a general introduction to EPIC architecture. There are other sources dedicated to that subject (Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0 available from http://developer.intel.com/design/itanium/index.htm). This discussion will instead cover details of the Itanium® 2 processor architecture that relate to software performance tuning using processor performance monitors. It is assumed in this discussion that you have a reasonable familiarity with the Itanium® Processor Family architecture and assembler.

A microprocessor runs an application by flowing the instructions into the execution units in synch with the arrival of the required data. When the flow is smooth, the execution is optimal as the processor never stalls waiting for either instructions or data. This chapter focusses on the architectural aspects of this synchronized flow.

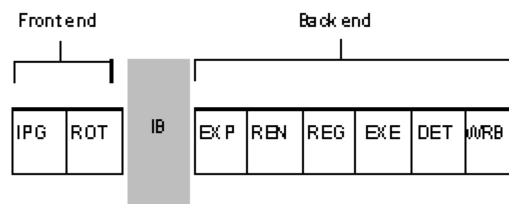
The flow of instructions and data into the functional units is controlled by the processor core pipeline and is at the heart of this kind of optimization. At the most upstream end, the instructions are read from the instruction cache or the Instruction Streaming Buffer (ISB), aligned for consumption by the pipeline back end. The back end first expands the bundle templates and disperses the instructions to the functional units. Next, any renaming activity of the registers is invoked and then the data is delivered to the functional units from the registers. With the data and instructions inplace the instructions can then be executed. The exception handling stage for branch mispredictions and exceptions are checked and then results are written back to the registers as required.

Analysis of the performance counters provides insight into the details of this process so you can tune your application for maximum performance.

Code Flow Restrictions and the Core Pipeline

The core pipeline organizes instructions and data so they can be issued to the functional units in a synchronized fashion. The 8-stage pipeline is divided into two pieces, a two stage front end and a six stage back end which are connected through an Instruction Buffer (IB) as shown in the following figure. The front and back ends of the pipeline operate asychronously with respect to each other.

Figure 3-1 Core pipeline



The front end generates the instruction pointers (IPG stage), and starts the L1 I-Cache and L1-ITLB accesses. It then formats the instruction stream (ROT stage) and loads the instruction buffer for consumption by the pipeline back end.

The back end consumes the instructions from the IB and organizes the required data to the appropriate registers and issues the instructions to the functional units in a 6 stage process.

Stage 1(EXP)

In the first stage (EXP), instruction templates are expanded and the dispersal is organized and issued.

Stage 2 (REN)

The REN stage handles the register renaming for register stack and register rotation manipulations. Instruction decoding is also done in this stage.

Stage 3 (REG)

The REG stage delivers data to the functional units either from the registers or by using bypasses for data generated by functional units for consumption by chained instructions. This stage also generates ay required spill or fill instructions required by the Register Stack Engine (RSE).

Stage 4 (EXE)

The EXE stage dispatches the instructions and data to the functional units requested by the instruction template. It also invokes the bypasses to provide output data from single cycle ALU instructions to the REG stage needed for upcoming instructions.

Stage 5 (DET)

The DET stage detects exceptions and branch mispredictions.This is where pipeline flushes due to exceptions or branch mispredictions are generated, causing the highest priority pipeline stalls. All potential exceptions are detected in this stage in time to prevent any write backs of the architectural state. This ensures maintaining the correct architectural state (for example, register contents). Stalls in the data delivery and floating point micropiplines that can result in core pipeline stalls are also detected at this stage.

Stage 6 (WRB)

Finally, the write back stage (WRB) writes the output to the appropriate output registers.

Back End Pipleine Stalls

The optimization methodology we will discuss in this guide concentrates on efficient instruction dispersal to the functional units, a process that is handled by the pipeline's back end. This means minimizing the cycles where the pipeline back end has stalled. There are three (back end) stages that can stall the back end of the pipeline. As the two halves of the core pipeline operate asynchronously, we will be concerned with only those front end stalls that succeed in stalling the pipeline back end. When we include the interactions with the micropipelines there are five large classes of stalls for the back end pipeline.

The back end pipeline stall conditions are prioritized with the priority increasing as the downstream end is approached. The highest priority stall occurs in the DET stage if there is an exception or branch misprediction detected. These will result in the pipeline being flushed making any activity further upstream irrelevant.

The pipeline back end interacts with several micropipelines which control the FPUs, Multi Media Units, the L1D and L2 cache accesses. The interactions with the FPU and L1D micropipelines can stall the pipeline back end in the DET stage but with lower priority than the exceptions or branch mispredictions.

The dominant cause of stalls in the EXE stage during application execution is dependency scoreboard stalls. This is the condition where data has not been delivered to a register in time for consumption by a functional unit. To ensure correct results the pipeline will stall until the data delivery has completed. The data delivery can be due to either a long latency instruction (floating point or multimedia) or a load which has not completed within the scheduled number of cycles.

The REN stage will stall the pipeline if there are an insufficient number of free general registers for it to satisfy any alloc instructions it must process. This will allow the REG stage to inject the required spill or fill instructions to invoke the register stack engine.

Finally when the front end has been unable to prepare a sufficient flow of instructions for the back end to keep the execution flow going, the back end will be spend cycles waiting for instructions and not making progress on the program's execution.

Memory Subsystem

The interaction of the code with the memory subsystem is usually the dominant part of the tuning process that results in restructuring the algorithm or data structures. The Itanium 2 processor has a three-level cache structure and in the Intel platform accesses main memory through the 870 chipset. Effective use of the memory subsystem requires that this interaction be optimized. A large part of this discussion will be focused on this part of the tuning process.

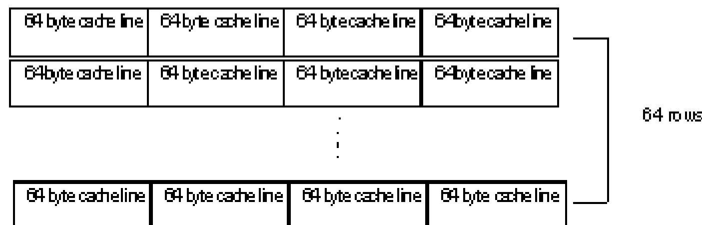
Most modern computers use a memory caching system. This allows for extremely fast access to frequently-used data and instructions. Data (and instructions) are loaded in a cache line which stores a sequential section of memory. An initial memory access request (load) causes not just the data requested be brought from memory but enough data to fill the cache line. The underlying assumption here is that if a program requires a particular piece of data, it is likely to require data in the adjacent addresses. A write to a memory location also causes the data in cache to be updated so that later use (input for subsequent instructions) has the correct value. In a multiprocessor system, a great deal of effort goes into the hardware design to ensure the coherency of the cache systems in the processors and thus the consistency of the data.

A memory cache is organized as a table with the rows corresponding to data stored in cache lines. The row that a particular piece of data (and the data in the associated cache line) gets placed in is determined by a subset of the bits in the address. For example, in a system with a 64-byte cache line size, the row is determined by truncating the bottom 6 bits of the address. Thus adjacent 64-byte strings of data are placed in adjacent rows. The number of columns in the table is the "associativity" of the cache. As the cache is not large enough to hold all data, new data requests force the replacement of cache lines. In order to use the cache system more efficiently, cache lines are grouped into associative sets, with the members of a set occupying a row of the table. When a new cache line has to be brought in, a sensible algorithm can be used to replace the element of the set which is least likely to be required again soon. Greater associativity results in more accurate replacement strategies and a greater efficiency for the cache.

The following figure illustrates a 4-way associative cache with 64- byte lines, like the L1 Data cache for the Itanium 2 processor. The number of rows would be the cache size/256bytes. For the L1D on the Itanium 2 processor, this would be 16KB/256B or 64 rows. The row index would be based on the base address of the cache line (i.e. the 64-byte aligned address just below the requested data). So the row index would be calculated from the address starting with the seventh

bit and extending to enough bits to cover all the rows. In the Itanium 2 Processor L1D cache the address cycle is (64*64) 4096 bytes, meaning that data with adresses separated by 4096 bytes will occupy the same associate set and could cause mutual cache line replacements.

Figure 3-2 Itanium 2 Processor L1 Data Cache



An added complexity to this structure is that a cache besides being organized in a set-associative table may be constructed as a banked structure as in the case of the L2 cache. This is usually done to facilitate the load and store ports into the processor core and manufacturing requirements. This in turn can cause restrictions to full bandwidth data access due to bank access conflicts.

When a request for data (load) can be satisfied by the cache, there is a "cache hit". This means the processor's cache structure and strategy served their purpose and a low latency for the memory access was achieved. When the cache does not have the data in residence at the time of the request, there is a "cache miss". Achieving the highest performance usually requires that the cache structure be incorporated into the algorithm and the data structures. The idea is to ensure the greatest efficiency in the cache usage. This in turn usually translates into minimizing cache misses, but the exact details of this become involved, particularly for a multi-layered cache structure as exists in the Itanium 2 processor.

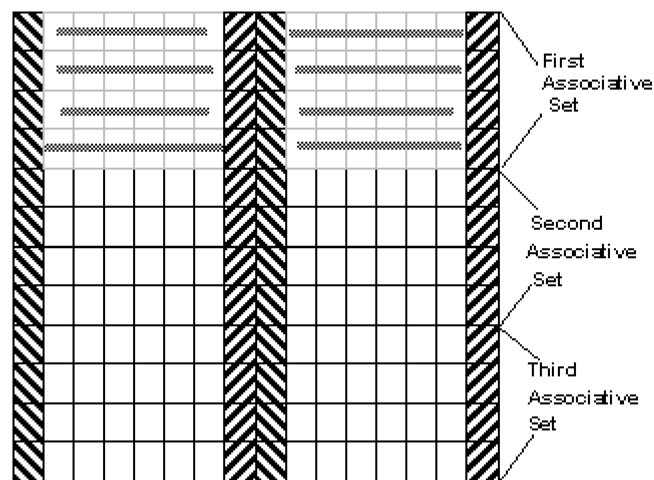
Itanium 2 Processor Cache Description

The highest speed part of the memory system is the L1 cache. There are separate level 1 caches for instructions (L1I) and data (L1D). Both are 16KB in size with 64-byte cache lines and have a *one cycle latency.* It is 4-way set associative so the bottom 12 bits (11:0) are used for cache indexing. The L1 data cache is only used for integer data. Studies have shown that floating point intensive applications need data samples that are too large to be placed in something that can be built with this speed. There are essentially no bank/port conflicts associated with L1 data cache access. It is a write through cache to ensure data currency. All integer loads (i.e. loads of general registers) go through this cache unless they are loads from uncacheable memory. Data access requests are initiated in order but the data can become visible out of order, meaning that the caches are

non-blocking and an earlier cache miss does not block the delivery of data from a subsequent cache hit. There are 2 load and 2 store ports connecting this cache to the register file, allowing up to 2 loads and 2 stores per clock cycle.

The L2 cache is unified with respect to instructions and data. It is 256 KB with an eight-way associativity and a 128 byte cache line size. This means the L2 cache has a total of (256KB/8X128 bytes) 256 associative sets or rows if displayed in the form shown in figure 2. Thus the L2 cache has an address cycle of 32KB. It is constructed of 16 banks each 16 bytes wide. The 16 bank, 8-way associative L2 unified cache is illustrated in the following diagram.

Figure 3-3 Physical cache representation with each of the elements of the associative sets (or "ways") being constructed of 8 of the 16 byte wide banks



Floating point data is loaded from this cache directly to the floating-point registers and is not loaded into the L1 cache. As with L1, data access requests are initiated in order but the data can become visible out-of-order. This is different than on Itanium processors where access was through a FIFO queue. The minimum L2 latency for integer and floating point loads are 5 and 6 cycles due to the different paths for integer and floating point data delivery. The complex queuing and bypass structure of this cache can cause the latency to be greater than these minimum values. All four memory ports can be used for loading floating point data from the L2 cache. This cache is made of 16 banks, each being 16 bytes wide and there are conditions that can cause bank access conflicts. Bank conflicts will be discussed in the chapters concerning memory access stalls and the BE_EXE_Bubble and BE_L1D_FPU_Bubble performance counters.

The L3 cache is again a unified cache. It is 1.5 or 3 MB in size and 6 or 12-way set associative respectively. The cache lines are 128 bytes. It has an eight entry, non-blocking queue. It is on die, delivering a maximum bandwidth of 32 GB/sec. Transfers to L2 (and the system bus) take four cycles so the latencies grow from the best case 12 cycles as the queue gets filled. Data accesses that also miss in L3 are escalated to the system bus and thereby to the chipset and main memory.

Memory Access and the L2 OzQ

The L1 data and L2 caches interact with the core pipeline and the register files through their own dedicated micropipelines. Stalls, cancels, redirects and the like can inhibit smooth flow of data into the register files for efficient use by the functional units and can cause core pipeline stall cycles. Of particular note is the L2 request queue that allows the L2 to service data access out of order.

All the caches on the Itanium 2 processor are non-blocking and can return data out-of-order. This is different than on the Itanium® processor where only the L1 data cache was non-blocking. With the Itanium 2 processor, L2 accesses are scheduled with an out-of-order queue called the OzQ. The OzQ has 32 entries. The OzQ handles the requests that cannot be satisfied by L1D. These include all stores, semaphores, floating-point data loads, uncacheable accesses, L1D load misses and L1D unresolved conflict cases. The L2 cache design requires less than 32 OzQ entries to hold the maximum number of L1D requests in conflict-free cases. Unfortunately, there are many conflict cases within the L2. These cases (bank conflicts, multiple misses to a single cache line, etc.) may increase request lifetimes within the OzQ. Thus, the additional entries allow the L1D pipeline (and FP loads) to continue to service hits and make additional requests of the L2 while the L2 resolves the conflicts. The conflicts increase the L2 latency and make L2 latency prediction in complex applications impossible.

It is possible for the L2 OzQ to become full due to the conflicts and unavailable resources. In these cases, the L1D pipeline will stall until the OzQ has satisfied some of its outstanding requests. This in turn will cause the core pipeline to stall as well.

The OzQ control logic allocates up to four contiguous entries per cycle starting from the last entry allocated the previous cycle (the tail). If there are too few entries available (between 4 and 12), the L1D pipeline stalls to prohibit any additional operations being passed to the L2. Requests are removed from the OzQ when they complete at the L2 when

- a store updates the data array (L2 Cache)
- a load returns correct data to the core
- an L2 miss request is accepted by the System Bus/L3

The OzQ control logic maintains three round-robin pointers to track head, tail, and issue. New requests are allocated at the tail pointer which indicates where the youngest OzQ operation exists. The head pointer is the oldest operation in the OzQ. The issue pointer is always between the head and tail and indicates where the issue logic should look for new operations to send down the L2

pipeline. A consequence of this head and tail organization is that holes may appear in the OzQ from operations that have issued (OzQ entries between the head and tail that are no longer valid). The OzQ is not compressed when these holes develop. Without compression, these holes are not available to new L1D or FP load/store requests. Thus, there may be instances where the OzQ control logic indicates that there is no more room for new L1D and FP requests, despite the fact that only a few OzQ entries are valid.

Stall cycles can develop due to the OzQ becoming full (aggravated by the holes just mentioned) and the pipeline stalls until it has cleared out sufficiently. When this happens, the order of data requests that can cause conflicts (multiple accesses to a single missing cache line, bank conflicts) need to be changed so that the conflicts and the associated OzQ recirculations and cancels are reduced. You can usually accomplish this by a simple reordering of source lines and explicit loop unrolling.

Cache misses can also cause entries to have long lifetimes in the OzQ. Use of prefetch intrinsics can reduce cache misses and reduce stall cycles due to the OzQ being full.

One additional side effect of the out-of-order L2/L3 data return is that synchronization code that actually relied on the blocking L2 cache on Itanium® processors may not synchronize as it did on Itanium 2 processors, causing runtime failures of the code. This is a result of incorrect coding. The EPIC architecture requires that the correct memory fencing semantics be followed, as outlined in the Intel® Software Developers Manual, Volume 2, Chapter 2.

Translation of Virtual Address to Physical Address

The translation of virtual address to physical address is required for the processor to access physical memory. This is done with a hierarchical system of a two-level Data Translation Lookaside Buffer and a Hardware Page Walker (HPW). When data accesses require updating these tables or invoking the page walker, considerable extra latency for the data access is incurred.

Translation Lookaside Buffers and the Hardware Page Walker

The two-level Data Translation Lookaside Buffers (DTLB) are part of the hardware chain that translates virtual addresses to physical addresses. They can be viewed as a 2 level cache structure for the virtual hash page table (which is accessed by the hardware page walker) in performing the translations to physical address. More detailed discussions of the virtual memory system can be found in Volume 2 of the Software Developers Manual.

An entry in the first-level data TLB, L1 DTLB, is required to get an L1 cache hit even for data that is resident in the L1 data cache. Consequently TLB misses cause stall cycles as they result in an L1 data cache miss and an L1 DTLB miss. The L1 DTLB is accessed in parallel with the L1 data cache and is part of the 1 cycle latency access system for integer data. Stores and floating point loads incur no penalty for a miss in the L1 DTLB.

The L1 DTLB has 32 entries, is fully associative, dual-ported and only supports 4KB pages. Larger pages are accommodated by multiple entries. When an L1 DTLB page translation is replaced, all of the associated entries in the L1 cache become invalid.

The second level DTLB (L2 DLTB) has 128 entries, is fully associative and quad-ported. It is accessed in parallel with the L1 DTLD and provides the protection information as well as the virtual to physical mapping. Integer stores and floating-point loads that miss the L1 DTLB incur no penalty. Integer loads that miss the L1 DTLB but hit the L2 DLTB, incur an 4 cycle penalty to transfer data from the L2 DLTB and an additional penalty to update the cache line.

The third level in the virtual to physical translation hierarchy is the Virtual Hash Page Table (VHPT) which is accessed by the Hardware Page Walker. When a L2 DLTB miss occurs, the HPW accesses the VHPT to locate the virtual to physical translation data and then updates the TLBs. The data access is then completed in a normal fashion.

The VHPT is maintained by the OS and and can reside in cacheable memory (OS specific). The location of the VHPT data pages must be provided to the HPW as they are needed. These locations can also be “cached” in the DTLB system so it can be accessed more quickly. If the locations of the VHPT data are not in the L2 DTLB then a VHPT fault is generated and the OS provides the translation to the HPW.

If the short format is used (as is the case with most OS’s) then each VHPT entry occupies 8 bytes so there are 1024 VHPT entries for each 8196 KB page of VHPT data. In order to locate the VHPT pages, one entry in the L2DTLB is required for each page of VPHT data that the HPW must access. If the VHPT data is cacheable then the HPW can update the DTLB system much more quickly than if the OS must access the data structure from main memory.

While there are 128 entries in the L2DTLB not all are available for accessing the application’s addressable space. The OS is allowed to reserve up to half the entries as translation registers but in practice you will find only a handful are reserved. In addition a certain number may be used for locating the VHPT. So depending on how many (distant) data pages are being accessed simultaneously, the number of available entries in the L2DTLB may be considerably less than the full 128.

If the required translation cannot be found in the TLBs, then an VHPT fault is generated which causes the OS handler to locate the VHPT. This is usually a very fast handler but still significantly slower than accesses handled completely by hardware.

Event-based Sampling with the VTune™ Performance Analyzer

4

Event-based sampling (EBS) with the VTune™ Performance Analyzer profiles all software executing on a computer based on the occurrence of microarchitectural events. These events are measured using the hardware performance counters on the processor. The data generated by EBS can be used to provide detailed information on the behavior of the microprocessor as it executes software. EBS captures the processor's execution context after a number of microarchitectural events have occurred. Some examples of microarchitectural events are branch mispredictions and cache misses. After the EBS data collection is complete, the VTune analyzer will display the events by process, thread, module, function, or line of code (when the code is compiled to generate debug symbols).

The number of events required to trigger a sampling of the execution context is called the sample after value. It is important to understand that samples are not meant to be collected after every occurrence of an event (every time a performance counter is incremented).The VTune analyzer can either automatically calculate the sample after value or the user can set the value manually in the GUI. When it automatically calculates the sample after value, an extra data collection run is required to determine this value. This process is called calibration. The VTune analyzer calibrates the sample after value so that an average of one sample is collected per millisecond.

After the VTune analyzer collects sampling data, it displays it in a bar graph or table view. The samples can be viewed by process, thread, module, function, or line of source. The last two items are only available if debug information is available. If debug information is not available the user can drill down to assembly code.

On Itanium 2 processors a sampled instruction program counter may be off by approximately 48 dynamic instructions. This means that in the source view, for Itanium 2 processors, the event actually occurs on an address before the address that the event actually appears on in the source view. This is called an event skid. The only events that do not suffer from event skid are event address register (EAR) events. Because of this, when using non EAR events, it is important to look at where the events occur in terms of blocks of code, not single lines. For example, consider the event data for an entire loop instead of just a single line of code or assembly in the loop.

The VTune analyzer is based on projects, Activities, and results. A project can have multiple Activities. An Activity has a specific data collector configuration and can have multiple results. Results within a project can be compared with each other by dragging and dropping them from the Project Navigator window on to another result's sampling data view.

The VTune analyzer also has remote sampling data collectors (RDC) for Windows* and Linux*. This allows the user to collect sampling data on a computer without the overhead of the VTune analyzer's GUI. On Windows, the RDC uses DCOM. The RDC for Linux uses a proprietary protocol over TCP/IP.

Only one instance of the VTune analyzer can collect sampling data on a computer at a time. There is also no request queuing mechanism in the RDCs, so only one user can profile a system with them at once.

A default EBS run collects data for the CPU_CYCLE (also known as Clockticks) and Instructions Retired. It computes the ratio of cycles per instruction retired. The events are consistently referred to by the short event names discussed in this manual and the more detailed Intel® Software Optimization Guide. Further all of the subcomponents accessed through programming the umask values (discussed in more detail in chapter 5) are predefined.

You can modify the choice of pre-selected events (Clockticks and Instructions Retired) when you configure the sampling collector using the EBS wizard. You can also select events after the first run by right-clicking the Activity in the Project Navigator window to bring up a menu and selecting "Modify Activity". It is usually best to create a copy of the Activity and modify the copy. That way all of your previous data continue to be available in the project. After you select a new set of events, click on the Start Activity button (green right arrow)to start the calibration and collection cycle for the new copy of the Activity. See the online help section "Working with Projects and Activities" for more information.

Limitations:

- You can collect sampling data on up to four events in a run, so calibration and data collection can result in a large number of application runs.
- At this time, the VTune analyzer interface is a 32- bit application. So collecting data for many events can create a very large load due to the OS changing the state of the processor between 32-bit and 64-bit modes for every data collection or calibration run. Collecting data on many counters (and consequently through many runs), may result in some degradation in accuracy. Using the remote collection mode results in a less intrusive data collection process and is recommended.

Performance Monitoring and Cycle Accounting

5

The objective of microarchitectural optimization is to maximize the flow of instructions through the CPU's functional units. This is equivalent to minimizing the CPU cycles where the core pipeline is stalled. To accomplish this in a methodical manner, the causes of pipeline stalls invoked by the execution of an application, must be categorized quantitatively. The dominant sources of pipeline stalls can then be remedied in an efficient manner.

On Itanium® processors, each stage that can stall the pipeline has an associated event that allows the accumulation of the stalls due to that stage. In addition, these events are prioritized so that in the event of multiple stall conditions, each stalled cycle is attributed to one and only one stage of the pipeline. Stalled CPU cycles are assigned to the highest priority pipeline stage experiencing a stall during that cycle. Priority of the stall increases as the end of the pipeline is approached. This creates a sum rule whose components divide the stalled CPU cycles between various architectural subsystems. These cycle accounting components correspond to different architectural features in the pipeline and in themselves categorize the dominant architectural features responsible for the execution inefficiency.

On Itanium processors, even cycles spent issuing instructions to the functional units invoke an event that can be explicitly counted. On the Itanium® 2 processors, cycles spent issuing instructions to the functional units must be calculated from other events.

The objective of microarchitectural performance tuning is to minimize the components of the sum rule that correspond to pipeline stalls. In other words, minimize cycles that are not spent issuing instructions to the functional units.

Cycle Accounting Sum Rule for Itanium 2 processors

The event, BACK_END_Bubble.ALL accumulates the cycles where the instruction pipeline stalled for any reason. The result of

```
CPU_Cycles - Back_End_Bubble.ALL
```

is equal to the number of cycles spent issuing instructions to the functional units. To determine the breakdown of the pipeline stalls, apply the main sum rule:

```
Back_End_Bubble.ALL =
 BE_Flush_Bubble
+ BE_L1D_FPU_Bubble
+ BE_EXE_Bubble
+ BE_RSE_Bubble
+ Back_End_Bubble.FE
```

The components are listed in decreasing priority (ie, from the downstream to upstream ends of the core pipeline). Each component accumulates stall cycles caused by different architectural subsystems. These are similar to the Itanium® processor performance monitoring events which work in a similar prioritized manner. The following table illustrates the comparison between Itanium 2 processor events and Itanium processor events used to count cycles lost in stalls.

Table 5-1 Comparison of Itanium 2 and Itanium processor events used for cycle accounting

Events for Itanium® 2 Processor Cycle Accounting	Events for Itanium® Processor Cycle Accounting
BE_Flush_Bubble	Pipeline_Backend_Flush_Cycle
BE_L1D_FPU_Bubble BE_EXE_Bubble	Data_Access_Cycle Dependency_Scoreboard_Cycle
BE_RSE_Bubble	RSE_Active_Cycle.d
Back_End_Bubble.FE	Unstalled_Backend_Cycle Inst_Access_Cycle Taken_Branch_Cycle.d

In order to compare the Itanium and Itanium 2 processor cycle accounting events equitably, you must group the events into sets corresponding to identical causes of the pipeline stalls. The Itanium 2 processor events BE_L1D_FPU_Bubble and BE_EXE_Bubble and the Itanium® processor events Data_Access_Cycle and Dependency_Scoreboard_Cycle form such a set. Between the two pairs of events, you can account for all memory access stalls and scoreboarded register dependency stalls On Itanium processors, the memory access stalls and scoreboard dependency stalls are assigned to individual counters, Data_Access_Cycle and Dependency_Scoreboard_Cycle respectively. On Itanium 2 processors, these stalls are assigned to counters more closely tied to the architectural subsystems.

The Itanium® processor counter RSE_Active_Cycle.d is a derived quantity and is equal to the difference of:

```
Memory_Cycle - Data_Access_Cycle
```

The intent of this guide is to present the counters and offer a methodical approach of their use in application optimization. The strength of the cycle accounting approach is that it makes evaluating the relative contributions of architectural usage inefficiencies very straight forward. You can start with the sum rule and then drill down into the largest components as required, to identify the sources of CPU cycles spent on anything other than executing instructions.

The cycle accounting sum rule components (events) have each subcomponents (subevents) which are accessed by selecting the appropriate umask (user mask). They are discussed in the technical documentation[1] in detail and are identified by a completer field added to the event name. For example, Back_End_Bubble would denote the parent event, with it's implied default umask value of 0. Back_End_Bubble.all is equivalent to the parent, with the ".all" completer explicitly designating the specific umask. The event Back_end_Bubble.FE, with the completer ".FE" is the subevent with the umask value of 1. In general, the subcomponents of the cycle accounting monitoring events do not have the same prioritized structure and the sum of the subevents may not rigorously be equal to the parent counter (which usually has a umask of 0). Subevents are discussed in greater detail in the chapters devoted to each of the sum rule components.

For Itanium performance events, you can set the umask through the VTune analyzer interface by selecting the event in the configuration menu, highlighting the selected event, and clicking the "Edit event" button. For Itanium 2 performance events, however, all of the subevents are explicitly predefined, making the process of selecting the umask unnecessary.

Analyzing the Application

The cycle accounting structure provides a methodical approach of analyzing an application's flow through the microarchitecture. It allows an immediate understanding of which architectural subcomponents are restricting the flow of instructions through the microprocessor's instruction pipeline by use of the primary sum rule. This feature is used to evaluate the origin of the important execution bottlenecks. The analysis process is a prioritized sequence through the components of the cycle accounting sum rule. In this guide, the discussion of the components is followed with a discussion of the most commonly occurring contributing sources of stall conditions.

The key to optimizing an application is to use performance monitoring events to identify the dominant stall contributions and their relative importance. Removing cache misses when they have a minimal contribution to the reduction in performance is an inefficient and wasteful exercise. Therefore it is critical that you prioritize the performance bottlenecks by their impact on stalling execution and attack them in the order of their importance.

While the sum rule shown above communicates this information, it is easier to quickly comprehend the impact when presented as a normalized ratio. The naïve thing to do is to normalize to CPU_CYCLES. This would be a zero sum game that makes it difficult to understand

1. See Itanium® 2 Processor Reference Manual for Software Development and Optimization

your progress. In fact, it is better in general, to normalize to a measure of the amount of work done (an application-dependent definition), or to Retired Itanium® Instructions (event code 8). For the purposes of the following discussion, the accumulated data will be normalized to the event IA64_Inst_Retired (IA64IR). This is a subevent of IA64_Tagged_Instructions_Retired and defined to count all instructions with a true predicate and all branch instructions regardless of predicate. It is therefore slightly different from the corresponding counter on Itanium® processors.

The application efficiency analysis expressed in Cycles Per Instruction (CPI) has a simple algebraic structure that enables you to drill down into any component of the stall cycle accounting. This is simplified by the denominator, IA64_Instructions_Retired (IA64IR) which is common for all the components and relatively stable even as the optimization progresses.

```
CPI = CPU_CYCLES/IA64IR
```

As you optimize the application algorithmically or with compiler flags, the total number of instructions (path length) changes. This in turn causes variations in CPI. However, it still remains one of the best normalizations for standardized (i.e. comparable) execution efficiency measurement.

The objective of microarchitectural optimization is the reduction of stall cycles. You can measure this reduction using the Back_End_Bubble event. If you define the quantity CYC_RET_INST as the number of cycles spent retiring instructions, then

```
CYC_RET_INST = CPU_CYCLES - Back_End_Bubble
```

This is equivalent to the Itanium® processor event All_Stops_Dispersed. Thus we can write:

```
CPI = CYC_RET_INST/IA64IR + Back_End_Bubble/IA64IR
```

and

```
Back_End_Bubble/IA64IR = ΣBE_Bubble_COMPONENTS/IA64IR
```

The BE_Bubble_Components (the elements in the cycle accounting sum rule) can be further broken down with the use of subevents and in some cases with models incorporating penalties for the occurrence of other architectural monitoring events (called “occurence events”, to be discussed a little later). This is particularly true for the memory access stalls that contribute to BE_EXE_Bubble and BE_L1D_FPU_Bubble. This leads to relations like:

```
BE_Bubble_Component/IA64IR ~
ΣOccurrence_Events*Penalty/IA64IR
```

where the sum extends over the architectural monitoring events (and in many cases differences between counts of such events) which have a relationship to contributing to the pipeline stalls accumulated in the corresponding BE_Bubble stall counter.

This last step is not as accurate as it was on Itanium processors due to the out-of-order data returns from the L2 cache and the complex scheduling that can result in the OzQ. The Itanium® processor L2 FIFO queue made such a modelling remarkably accurate. On Itanium 2 processors, the

out-of-order returns from the OzQ allow access penalties to mask each other, making things more complex. However, you can still use memory access modelling as a guide to help determine where the most benefit is likely to be gained.

Cycle Accounting Components

This section discusses the components of the sum rule at a high-level. Chapters 6 through 10 are dedicated to a more detailed discussion of the contributions to each of these components.

BE_Flush_Bubble

This counter accumulates cycles lost due to stalls caused by the most downstream stage of the processor pipeline that can cause pipeline stalls, the DET stage. The WRB stage cannot stall. There are two contributing sources, which are prioritized:

- pipeline flushes due to exceptions (be_flush_bubble.xpn which has a umask of bxx10)
- pipeline flushes due to branch mispredictions (be_flush_bubble.bru which has a umask of bxx01).

Pipeline flushes due to exceptions and branch mispredictions can sometimes be associated with front-end pipeline stalls. The reason for this is that when the compiler generates the code, it makes assumptions about the code flow, hence the branch misprediction/exception. It also places the expected code blocks together in the binary for efficient access and storage in the instruction cache. Thus a pipeline flush can also result in a search for an infrequently used code block resulting in I-Cache misses and a stall in the front end of the pipeline. Chapter 8 explains this counter in greater detail.

BE_L1D_FPU_Bubble

The BE_L1D_FPU_Bubble counter accumulates cycles stalled in the DET stage of the core pipline due to stalls in the L1D and FPU micropipelines. In practice, there are usually few stall cycles due to the FPU micropipeline. Stalls due to the L1D micropipeline are caused by congested data flow and DTLB misses and HPW activity. Thus some of the memory access stalls are recorded in this counter. Chapter 7 explains this counter in greater detail

BE_EXE_Bubble

The BE_EXE_Bubble counter accumulates stall cycles in the EXE stage of the pipeline. These include stalls due to memory access latencies and scoreboard dependency stalls. The bulk of the memory access stalls are recorded here. Chapter 6 explains this counter in greater detail

BE_RSE_Bubble

The BE_RSE_Bubble counter accumulates stalls that occur in the REN stage. It can stall if the available resources are insufficient. This causes the Register Stack Engine (RSE) to invoke the backing store mechanism to free up physical registers to satisfy the general register stack requirements. The REG stage of the core pipeline invokes the register stack engine by injecting the required RSE loads and stores. Recompiling with profile guided feedback (Qprof_gen/Qprof_use) and interprocedural inlining (Qipo) can usually have significant impact on register usage. Alternatively, reducing the number of arguments in heavily used function chains through the use of structures and the reduction of dependency on recursive algorithms can reduce register allocation pressures. Chapter 9 explains this counter in greater detail

Back_End_Bubble.FE

The Back_End_Bubble.FE counter accumulates the stall cycles due to the the back end pipleine stalling due to a lack of instructions for dispersal. If the front end of the pipeline does not provide instructions in time, it causes core pipeline back end stall cycles. This means that none of the stalls discussed above were present as they are all higher priority. The typical cause for these stalls is the compiler not grouping the hot instruction blocks together properly and the L1 I-Cache being used inefficiently. This can of course be the result of branch mispredictions as mentioned before, but can occur even when the dynamic branch prediction hardware is correctly predicting branches. Recompiling with profile guided feedback (Qprof_gen/Qprof_use) and interprocedural inlining (Qipo) can usually have significant impact on these kinds of stalls. Chapter 10 explains this counter in greater detail

Contributions to Core Pipeline Stalls

Cycle Accounting and Occurrence Events

The cycle accounting capability in the Itanium® Processor Family is based on counting the cycles where the core pipeline is stalled. The stall is assigned to the most downstream (highest priority) stage that registered a stall condition on that cycle. You can investigate the underlying reasons for the stalled cycles with the use of the other microarchitectural event counters, occurrence events, that accumulate as specific architectural interactions are encountered.

As an example, consider an integer data load that misses in the L1 data cache but that can be satisfied from L2. The architectural event is an L1 Data cache miss which causes the L1D_READ_MISSES occurrance event count to be incremented by one. The increased latency for the load, however, may cause the pipeline to stall in the EXE stage due to a dependency on the availability of the data. The prioritized counter, BE_EXE_Bubble is incremented by one for each cycle the pipeline is stalled until the data is delivered and available to the functional unit. This assumes that there were no pipeline flushes or micropipeline stalls during this time due to an

exception or branch mispredict, DTLB miss etc. Those would cause a stall condition in the more downstream pipeline DET stage, and the cycles would then have been assigned to the higher priority counter (BE_FLUSH_Bubble or BE_L1D_FPU_Bubble).

While a given stalled cycle is assigned to only a single prioritized component of the cycle accounting performance monitors, a large number of the occurrance event counters could be incremented on a single cycle. This adds complexity to interpreting the occurrance event data and requires that you take care when attempting to do so. As the EPIC architecture allows out-of-order memory returns while the L1 data cache miss was occurring in the example just discussed, another load that required access to main memory could have been occurring in parallel, causing cache misses in L1D, L2 or L3 during the same cycle.

Occurrence Events

The presentation of other relevant counters is organized under the prioritized stall cycle counters that the associated occurrence events contribute to. This allows a discussion of how they contribute to the stalled cycle count. More importantly it groups events under a structure that mirrors the EBS investigation process.

To determine stall cycle penalties induced by individual occurrence events, you must construct the test procedure very carefully. In general, construct the test cases in assembler because compilers work very hard at hiding latencies and making effective use of pipelines.

In the course of discussing occurrence events, some of the test cases used will illustrate what coding structures can induce stall conditions. The test cases will also explicitly show how to measure the stall penalties.

Subevents

Almost all performance monitoring counters are subdivided into more specific components through the use of subevents. Each event has a unique event code and the subcomponents are specified by further specifying a (non-zero) user mask (umask). A zero value for the user mask (usually) collects data for all subevents simultaneously. If several subevents fire during a given cycle, the parent event (umask=0) may only be incremented by one count. This means that exact sum rules cannot always be constructed. However, in general, the approximation that the sum of the components is equal to the parent counter usually works well enough to arrive at the required understanding of the program execution. Note that this statement is also true for the cycle accounting events. The sum of the subcomponents (umask != 0) is not guaranteed to be equal to the value of the parent counter (umask=0). It should be noted that for many events, the sum of the subevents do exactly equal the parent. For example, the counter L2_Data_References.L2_All is the parent (even though the umask is 3) and is equal to the sum of L2_Data_References.L2_Data_Reads (umask 0x01) and L2_Data_References.L2_Data_Writes (umask 0x02).

As most analyses are performed with the VTune™ Performance Analyzer, which does statistical sampling, the exact vs non-exact nature is usually within the statistical accuracy anyway. The default umask value can be changed through the VTune analyzer graphical user interface by selecting the event highlighting it in the selected event window and clicking the "Edit event" button. In the Itanium 2 processor pack for the VTune analyzer, this procedure is not necessary as all of the subevents are preprogrammed with the specific umask for the users' convenience.

Event Skid and EAR Events

During data collection mode, when the VTune analyzer takes an interrupt to record the Instruction Pointer (IP) associated with the occurrence of the triggering event, there can be some inaccuracy in recording the IP. This is called an event skid and results in events being associated with slightly incorrect locations in the program. It is usually not a big problem but if very high location precision is required, there are a class of performance events called EAR events (exact address register) where the hardware records the IP when the event occurs. There are 6 such types of events that can be programmed for data collection, L1 instruction and ITLB misses, L1 data and DTLB misses, ALAT misses and front end stalls.

Studying the Architecture with Microbenchmarks

Microbenchmarks are sample programs used to create "atomic" test cases of a highly controlled type. They allow individual architectural events to be induced in repetitive loops so that the stall cycle penalties for the associated occurrance events can be precisely measured. Microbenchmarks have the added benefit in that you can use them to study, in a controlled manner, which events and subevents are incremented under precise circumstances. This understanding can then be extrapolated back to analyzing the real application. However this extrapolation introduces a great uncertainty in that the real application and the microbenchmarks may interact with the complex architecture and queing in very different manners. That said, the microbenchmarks are a good starting point for such an analysis and serve as the practical basis of learning microarchitectural tuning techniques using event-based sampling.

Dominant Sources of Execution Inefficiency

The remainder of this guide is organized in chapters devoted to the five components (or events) of the cycle accounting sum rule. They are discussed in an order that reflects the most likely dominant contributions to the execution inefficiency that an application might encounter. Memory access stalls are usually what is encountered so most of the first two chapters, devoted to BE_EXE_Bubble and BE_L1D_FPU_Bubble, explain this kind of stall. The remaining chapters deal with cycles lost to pipeline flushes, front end stalls, and RSE activity.

Data Collection Restrictions

As stated in chapter 4, data can be collected on at most four events during a single data collection run. This is because there are 4 programable monitoring units (PMUs) that can count the signals associated with the event occurrence. Further there are restrictions on which events can be collected simultaneously.

L1 Data Cache Event Restrictions

Due to hardware limitations, certain L1D cache events share lines with each other and with certain other events. Consequently, there are combinations of events that cannot be used simultaneously and must instead be used in separate data collection runs. The L1D cache events are divided into five sets. Events within a set can be taken within a single data collection run. Events from different sets cannot. For example, the events L1DTLB_Transfer and L2DTLB_Misses, which count how frequently the data translation lookahead buffers do not have the needed information are in Set 0. BE_L1D_FPU_Bubble, which counts (among other things) the cycles lost due to DLTB misses, is in Set 1 and cannot be sampled in the same data collection run as the DLTB events in Set 0. The VTune analyzer is aware of the L1D data sets and will automatically take multiple runs as required. There are no constraints on the use of umask values associated with data collection of L1D events. The L1D_READ and DATA_REFERENCES events in set0 and set1 are functionally identical events that can be used in conjunction with set 0 or set 1 events.

The five sets are listed in the following tables :

Table 5-2 Set 0 events

Event Name	Event Description
L1DTLB_TRANSFER	L1DTLB misses that hit in the L2DTLB for accesses counted in L1D_READS
L2DTLB_MISSES	L2DTLB Misses
L1D_READS_SET0	L1 Data Cache Reads
DATA_REFERENCES_SET0	Data Memory references issued to the memory pipeline

Table 5-3 Set 1 events

Event Name	Event Description
L1D_READS_SET1	L1 Data Cache Reads
DATA_REFERENCES_SET1	Data Memory references issued to the memory pipeline
L1D_READ_MISSES	L1 Data Cache Read Misses

Table 5-4 Set 2 Events

Event Name	Event Description
BE_L1D_FPU_Bubble	Accumulated pipeline stall cycles due to L1 Data cache or FPU micro pipelines

Table 5-5 Set 3 Events

Event Name	Event Description
Loads Retired	Loads Retired
MISALIGNED_LOADS_RETIRED	Retired Misaligned loads
UC_LOADS_RETIRED	Retired Uncacheable Loads

Table 5-6 Set 4 Events

Event Name	Event Description
MISALIGNED_STORES_RETIRED	Retired Misaligned Stores
STORES_RETIRED	Retired Stores
UC_STORES_RETIRED	Retired Uncacheable Stores

L2 Data Cache Event Restrictions

L2 cache events are divided into 6 sets. Only events within a set (or non-L2 events) can be measured at the same time. Each set is selected by the event code programmed into PMC4 (i.e. if you want to measure any of the events in this set, one of them needs to be measured by PMD4). Within a set, certain events can only be measured by PMD4. There may also be some limitations on umasks in which the prime event (the L2 event using PMD4) dictates the umask for certain companion L2 events. These are noted by set. Monitors belonging to each set are explicitly presented in the following sections.

L2 Cache events (set 0)

Either one of the L2_OZQ_CANCELS* events or L2_IFET_CANCELS must be measured by PMD4. These events use the same umask. Only 1 of the 3 L2_OZQ_CANCELS* events can be measured at any one time.

Table 5-7 Performance Monitors For L2 Cache Set 0

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_IFET_CANCELS	0xa1,0xa5,0xa9,0xad	1	Instruction fetch cancels by the L2.
L2_OZQ_ACQUIRE	0xa2,0xa6,0xaa,0xae	1	Clocks with acquire ordering attribute existed in L2 OZQ
L2_OZQ_CANCELS0	0xa0	4	L2 OZQ cancels
L2_OZQ_CANCELS1	0xac	4	L2 OZQ cancels
L2_OZQ_CANCELS2	0xa8	4	L2 OZQ cancels
L2_OZQ_RELEASE	0xa3,0xa7,0xab,0xaf	1	Clocks with release ordering attribute existed in L2 OZQ

L2 Cache events (set 1)

L2_L3ACCESS_CANCEL must be measured by PMD4.

Table 5-8 Performance Monitors For L2 Cache Set 1

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_DATA_REFERENCES	0xb2	4	Data read/write access to L2
L2_L3ACCESS_CANCEL	0xb0	1	Canceled L3 accesses
L2_REFERENCES	0xb1	4	Requests made from L2

L2 Cache events (set 2)

L2_FORCE_RECIRC must be measured by PMD4.

Table 5-9 Performance Monitors For L2 Cache Set 2

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_FORCE_RECIRC	0xb4	4	Forced recirculates
L2_ISSUED_RECIRC_OZQ_ACC	0xb5	1	Count number of times a recirculate issue was attempted and not preempted
L2_GOT_RECIRC_OZQ_ACC	0xb6	1	Counts number of OZQ accesses recirculated back to L1D
L2_SYNTH_PROBE	0xb7	1	Synthesized Probe

L2 Cache events (set 3)

L2_Bad_Lines_Selected, L2_Bypass, and L2_Store_Hit_Shared share the same umask.

Table 5-10 Performance Monitors For L2 Cache Set 3

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_BAD_LINES_SELECTED	0xb9	4	Valid line replaced when invalid line is available
L2_BYPASS	0xb8	1	Count bypass
L2_STORE_HIT_SHARED	0xba	2	Store hit a shared line

L2 Cache events (set 4)

Either one of L2_OPS_Issued, L2_Issued_RECIRC_IFETCH, or L2_GOT_RECIRC_IFETCH must be measured by PMD4. These three events share the same umask.

Table 5-11 Performance Monitors For L2 Cache Set 4

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_GOT_RECIRC_IFETCH	0xba	1	Instruction fetch recirculates received by L2D

Table 5-11 Performance Monitors For L2 Cache Set 4

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_ISSUED_RECIRC_IFETCH	0xb9	1	Instruction fetch recirculates issued by L2D
L2_OPS_ISSUED	0xb8	4	Different operations issued by L2D

L2 Cache events (set 5)

Either one of L2_OZQ_FULL, L2_OZDB_FULL, L2_VICTIMB_FULL, or L2_FILLB_FULL must be measured by PMD4. These four events share the same umask.

Table 5-12 Performance Monitors For L2 Cache Set 5

Symbol Name	Event Code	Max Inc/Cyc	Description
L2_OZQ_FULL	0xbc	1	L2D OZQ is full
L2_OZDB_FULL	0xbd	1	L2D OZ data buffer is full
L2_VICTIMB_FULL	0xbe	1	L2D victim buffer is full
L2_FILLB_FULL	0xbf	1	L2D Fill buffer is full

BE_EXE_Bubble for Memory Access Stalls in the EXE Pipeline Stage

6

Stall cycles in the EXE stage of the pipeline occur in applications (mostly) when execution units do not find the required data in the registers allocated for the operation. Data dependency stalls occur for the following reasons:

- Data has not yet arrived from the memory subsystem (cache or main memory). This is a memory access stall.
- A previous functional unit not has not completed its final writing of the result. The latency of the functional unit computing a result has not been absorbed completely by intervening scheduled instructions. This is a functional unit latency stall.

The compiler tries to schedule instructions so these stalls do not occur. But due to finite cache and memory size, data may not be available in the cache system the compiler targeted. Also if there are computationally intensive sections of the source, it may not be possible to hide the functional unit latency.

This chapter discusses:

- the specific sources of EXE stage pipeline stalls
- the architectural features involved in a stall
- how to identify the causes and relative importance of these stalls

With this information, you can consider and systematically evaluate the optimization strategies to remove the performance bottlenecks that caused the pipeline to stall.

Memory Access Stalls

Memory access stalls occur when the data is not available in the caches as expected. The instructions that are dependent on this data being loaded and available, will stall until the load has completed. There are two main causes for the data not being available:

- The data is not resident in the desired cache level (cache miss)
- The virtual to physical translation did not occur optimally (DTLB miss)

A DTLB miss will result in a cache miss after the translation has been accomplished. A level 1 DTLB hit is required for integer data retreived from the L1 Data cache, a level 2 DTLB hit is required for floating point data to be retreived from the L2 cache. These kinds of memory access stalls will be discussed in chapter 7 as the resulting stall cycles are accumulated by both BE_EXE_BUBBLE and BE_L1D_FPU_BUBBLE.

Compilers schedule instructions assuming the following optimal latencies for data loads:

- integer data is loaded from the L1 cache with a one cycle latency
- floating point data is loaded from the L2 cache with a 6 cycle latency

The compiler may schedule instructions so as to absorb more latency than the minimum but will usually be able to absorb at least the minima stated above.

There are basically two conditions that can cause the delivery of data to take more clock cycles than the optimum that the compiler uses for minimum latency scheduling:

Cache misses

These occur when data is not in the desired cache and data retrieval requires access to a slower cache, memory, or even disk.

Data address conflicts

These stalls can occur when multiple data accesses are issued on a single cycle or in rapid succession. Address conflicts can cause the data deliveries to interfere with each other. Access paths for integers and floating point (FP) data are different, so address conflicts for these data types differ.

Optimizations to Remove Memory Access Stalls

Always focus your efforts on reducing the largest contributions to the stall cycles and do not work on issues that make little difference to the overall performance. Before addressing the issue of reducing memory access stall cycles, first establish that they contribute in a significant way to the performance of your program.

The following are some general suggestions for removing memory access stalls. Many of these suggestions are independent of the computer architecture. Both this chapter and chapter seven on stalled cycles accumulated in the BE_L1D_FPU_Bubble event, are mainly concerned with identifying the sources of memory access stalls. Once the exact nature of the stall is identified, you can formulate an appropriate resolution.

Cache Misses

In general, if cache misses cause a significant number of stall cycles, the data needs to be prefetched in advance to ensure availability or a more localized data use is required. This can mean:

- Raise the optimization level passed to the compiler
- Inserting prefetch intrinsics into the source appropriately
- Restructuring the algorithm to do more work on a given piece of data.
- Restructuring the data to make sequential accesses use sequential addresses (i.e. same cache line). Cache misses occur because the data is either not being re-used efficiently or the data is not organized to take advantage of the whole cache line concept.

A common occurrence is for data to be organized as a linked list of structures. The program then walks through the linked list. Probably the best solution for this is to allocate space for a block of structures and organize the data within the block as arrays. The advantage is that structure elements will be stored consecutively allowing very efficient use of the cache lines. Further, while the current block is being analyzed, the next block in the linked list can be prefetched and the memory access latency completely hidden.

Frequently, very little of the data in the structure is used by the algorithm at a given time. This results in there being little advantage in loading a cache line, as only a small fraction of the data contained in the cache line is utilized.

From the perspective of using the cache efficiently, it is better to have a structure of arrays than a linked list of structures. If this is not practical, keep floating point and integer data in separate parallel data structures. This will help in utilizing the cache line structures in the L1 (integer only) and L2 caches more effectively.

Alternatively, if the code reuses a small block of data but only after walking through very large blocks (that are not used), the cache has to be reloaded each time the program returns to working with the small block of data. In such a case, it is better to restructure the algorithm if possible, to completely finish working with a piece of data before moving on. This is really the usage model for which the cache line concept is intended.

Another option is possible if the data resides in L3 but not in the high speed cache that the hardware uses for access. In the case of looping code, the memory access latency is absorbed through pipelining and rotating registers. By unrolling the loops aggressively, you can increase the cycles per iteration of the loop and also increase the amount of latency absorbed by the pipelining. If you can do this to the level that the compiler's optimization automatically absorbs the L3 latency, i.e. 13 cycles, (see the discussion that follows) then the cache misses will not contribute to any pipeline stalls. Keep this option in mind while optimizing applications running on Itanium® Processor Family architectures.

Address Conflicts

Address conflicts are more common than most people realize and can be as costly as cache misses. Address conflicts are caused by technical details of the cache and cache access hardware and are therefore more difficult to understand, though sometimes much easier to avoid.

To identify such access conflicts, use the VTune analyzer's event based sampling feature to determine if and where address conflicts are a problem, then use a debugger to identify the variables that have the address conflicts.

Address conflicts occur because of features in the access mechanisms used by one or more of the caches. In the Itanium® 2 processor there are two main types of address conflicts, bank conflicts and secondary misses to an L2 cache line.

Bank Conflicts

The L2 cache is made of 16 banks, each 16 bytes wide. If two (more common with floating point) loads try to access the same bank on the same cycle, one of the accesses is forced to be re-issued by the L2 OzQ. This, for example, causes the floating point latency of the access for the second load to change from 6 cycles (when the two accesses go to two different banks) to 12 cycles.

There are two types of mechanisms that can create bank conflicts:

- An access conflict between two elements contained in the same cache line.
- An access conflict between two blocks of data (arrays) which have a base address alignment and a coherent access pattern.

The second of these can become a significant issue when looping code accesses multiple arrays. However, you can check these alignments easily.

If a linked list of structures creates an alignment problem, it may be a little more difficult to fix as it may not occur on every access. This illustrates another complexity associated with linked lists, in that the alignment of the elements can be made excessively complicated, particularly if they are dynamically managed. This kind of coding practice may seem elegant and save space but it makes it extremely difficult for the compiler to generate code that efficiently navigates the data and harder still for the hardware to bring it to the functional units in a timely manner.

As an example of an address conflict, consider floating point access on Itanium® 2 processors. Floating point data is loaded from the L2 cache directly to the floating point register files. If there are two load instructions for two arrays issued on the same clock cycle to addresses that map to the same bank, one of the resulting entries in the OzQ is re-issued. This causes the latency for the access for both loads to change from 6 cycles (for two accesses to two different banks) to 12 cycles, where this is determined by the re-issued OzQ entry.

Once you identify an addressing conflict of this type, it is quite straightforward to fix the problem.

Example:

The following line of code creates a 256-byte aligned buffer,

```
buf = buf + 256 - ((UINT64)buf%256);
```

where UINT64 is simply a typedef'd unsigned 64-bit integer, i.e. a pointer.

The address can be incremented by whatever multiple of 16 bytes that will resolve the bank conflict. Note that you should always keep buffers 16-byte aligned to allow the option of double loads. Malloc returns a 16-byte aligned address.

In a real application, cache line replacements occur regularly. This replacement rearranges the bank allocations with respect to addressing. The 16 L2 banks correspond to two L2 cache lines. The pattern repeats as four sets to give the complete 8- way associative set.

The simple address conflicts illustrated with microbenchmarks in this document should not give the impression that these effects are always predetermined. In an application that is algorithmically defined, by which is meant the code execution is mostly independent of the data values, then the relative data alignments in cache is rather straightforward. In an application that is strongly driven by data values, however, this may not be the case and the relative alignments of the data within the bank structure of the caches will be more difficult to predict.

As a general comment, if a source line shows a lot of stall cycles accumulated in BE_EXE_Bubble, you should check the bank conflict and recirculation occurrence events to see if they represent a significant fraction of the data accesses associated with those source lines. If they do, it is worth the time to bring the application up in a debugger to look at the addresses of the variables (source and target).

There are occurrence events on Itanium® 2 processors that identify some address conflicts. With the VTune analyzer, you can use the event, L2_OzQ_Cancels1.Bank_Conflict to explicitly count bank conflicts. After collecting the data, you can drill down to the source and assembly code level to identify the location of the bank conflicts.

Secondary L2 Cache Misses

The OzQ will recirculate data accesses if a pending request for the same cache line caused an L2 cache miss. Only one pending access to a missing cache line in L2 is escalated to L3 and/or the system bus at a time. Subsequent misses that occur while the pending line is being updated recirculate until the cache line has been completely updated. While the first miss can return data from the first part of the cache line that is loaded to the L2 cache, any further accesses to that cache line prior to its complete replacement must wait until the cache line has been completely updated. This causes an extra latency in the data delivery to the registers.

There are several techniques to avoid the extra latency. Certainly the best is to avoid the cache miss latency entirely by either

- building the program with /O3 optimization to generate the prefetches
- using prefetch intrinsics

Alternatively, the secondary miss can be avoided by

- sepperating the source lines that access the same structure or array sufficiently to allow the cache line to update from the first access. This will take about 10 cycles if the data is in L3. Note: this approach really isn't possible if the data is in main memory.
- Alternatively replacing arrays (or linked lists) of structures with structures of arrays will avoid putting the elements in the same cache line.

Counting recirculations due to secondary L2 misses cannot usually be done exactly on Itanium® 2 processors. An upper limit on the number can be determined by summing four sub events of L2_FORCE_RECIRC. The sum may double count some secondary misses and even include some misses to multiple cache lines from the same associative set. This will be discussed in some detail at the end of this chapter.

Functional Unit Stalls

Computers require functional unit latency stalls to ensure results are computed correctly. In a chain of instructions, the output of one instruction may be used as the input of another. If there are an insufficient number of instructions (cycles) between the two to absorb the operational latency of generating the data, the second operation must stall until the data is ready. This is called a "scoreboarded" stall. Many of the integer operations on the Itanium® 2 processor have single cycle latencies. As one cycle is required between a "generating" instruction and a "consuming" instruction to avoid the "read after write" dependency violation, a one cycle instruction never causes a functional unit stall. The most likely sources of these kinds of stall cycles come from Multi-Media (MM) integer instructions or floating point operations.

For integer instructions, the functional unit stalls are explictly measured by the event BE_EXE_Bubble.GRGR which counts a subset of the cycles accumulated by BE_EXE_Bubble.GRALL. So identification of this as an issue in the execution inefficiency is straightforward. It is most likely to appear in a chain of MM instructions coded with instrinsics. The solution is to interleave other instructions in between to allow the latencies to be absorbed.

Floating point intensive applications are more likely to encounter functional unit stalls. This is due to the typical longer latency of the basic floating point instructions used to build up complex calculations. It is more difficult to uniquely identify functional unit stalls in floating point calculations than for integers. While BE_EXE_Bubble.FRALL counts both floating point memory access stalls as well as functional unit latency stalls, there is no corresponding .FRFR subevent to distinguish them. To verify what is happening, look at the disassembly listing of the code in the source view displayed by the VTune analyzer. If after analyzing the code, you find that functional unit latency is the problem, then reorganize the calculations to absorb the latency. This means reducing the number of divisions and square roots (particularly long latency operations) by collecting intermediate results and regrouping mathematical expressions. The use of lookup tables and interpolation calculations is probably the best way to improve the performance in such a case,

if the required accuracy can be maintained. Large performance improvements can be gained by this strategy in addition to reducing stall cycles. With this strategy, however, there is a trade-off of space for speed.

BE_EXE Bubble

Most memory access stall cycles are accumulated by the BE_EXE_Bubble counter. This counter accumulates stall cycles in the EXE stage of the pipeline. These stall cycles occur mostly because the data loaded into registers are not ready for consumption by the functional units. These dependency stalls break down to two sources:

- Data has not arrived from the memory subsystem in time for use by the functional unit
- Data from a functional unit was not written back to the register in time for use by a subsequent instruction.

In both cases, the problem is insufficient independent instructions between the placing of the data in the register (load or instruction output) and its later use as input for other instructions, to absorb the latency of getting the data into the register.

You can analyze the subevents of the BE_EXE_Bubble counter to gain a clearer understanding of the kind of memory accesses causing the pipeline to stall.

The following table shows the subevents for BE_EXE_Bubble that can be selected with a specific umask value. In the VTune analyzer, the subevents are predefined with the specific umask values.

Table 6-13 BE_EXE_Bubble subevents

Extension	PMC.umask	Description
ALL	B0000	Back-end was stalled by exe
GRALL	B0001	Back-end was stalled by exe due to GR/GR or GR/load dependency
FRALL	B0010	Back-end was stalled by exe due to FR/FR or FR/load dependency
PR	B0011	Back-end was stalled by exe due to PR dependency
ARCR	B0100	Back-end was stalled by exe due to AR or CR dependency
GRGR	B0101	Back-end was stalled by exe due to GR/GR dependency
CANCEL	B0110	Back-end was stalled by exe due to a canceled load
BANK_SWITCH	B0111	Back-end was stalled by exe due to bank switching.
ARCR_PR_CANCEL_BANK	B1000	ARCR, PR, CANCEL or BANK_SWITCH
---	B1001-b1111	(* nothing will be counted *)

In the case of general registers, the two classes of dependency stalls can be explicitly measured:

- BE_EXE_Bubble.GRALL accumulates all integer data dependency stalls and
- BE_EXE_Bubble.GRGR accumulates stall cycles due to integer functional unit latencies not being completely absorbed by the instruction scheduling

Using the subevents, the stall cycles attributed to loading integer data can be approximated from the difference, as:

```
BE_EXE_Bubble.GRALL- BE_EXE_Bubble.GRGR
```

Frequently just collecting data on BE_EXE_Bubble.GRALL is sufficient as the compiler usually hides the functional unit latency. The difference is not exact because these two counters are not prioritized and there are situations where both could increment on a single cycle. For example a load and a MM instruction issued on the same cycle and the load misses in L1. If the code tries to use both results too soon then both of the stall conditions would be true but the functional unit stall would mask the memory access stall.

Since there is no corresponding pair of subevents of this type for floating point loads, there is some uncertainty in measuring the stall cycles due to floating point data access. If only the performance events can be used, you must also look at the associated memory subsystem

occurrance events (besides BE_EXE_Bubble.Frall) to determine the severity of floating point data access stalls. With the VTune analyzer you can simply drill down to the disassembly view and code inspection should make it obvious.

The rest of this chapter will focus on memory access stall cycles that contribute to BE_EXE_Bubble.GRALL-BE_EXE_Bubble.GRGR and BE_EXE_Bubble.FRALL. It is assumed that these are the dominant sources of stall cycles counted in the EXE stage. This pipeline stage, of course, also accumulates stall cycles due to functional unit latency (E.g. BE_EXE_Bubble.GRGR which counts integer functional unit latencies) that the compiler did not absorb through scheduling. If there is a significant fraction of stall cycles (or contribution to CPI) due to functional unit latencies, you must interleave a greater degree of memory access oriented code with the computations.This gives the compiler more flexibility in scheduling and should reduce this stall cycle source.This however, may be very difficult to actually accomplish.

Memory Access Latency Penalties

In order to evaluate the relative importance of various types of memory access stalls, you need to determine the individual penalties for each type of stall. The major sources of such stalls are:

- Cache misses where data is recovered from a more distant (longer latency) part of the memory subsystem. As the compiler usually schedules for the highest speed cache, the EXE stage will stall for the difference of the latencies.
- DTLB misses (discussed in the next chapter).
- Address conflicts causing data access hardware to execute more complex access patterns thus increasing the effective latency of the data access.

Equipped with this knowledge, you can evaluate the relative contributions of the memory access stalls by weighting the appropriate occurrence event counters by their associated penalties as an approximation to reconstruct the cycle accounting. Due to the out-of-order nature of data returns from the L2 cache, this model for cycle accounting has larger uncertainties than it would on Itanium® processors.Further an assumption must be made about how much latency the compiler scheduling absorbs. The approximate relations that we determine in this chapter can only be used as upper limits to the stall cycles associated with the occurrance events. They do serve as a guide in evaluating what effects are impacting the execution efficiency.

Memory Access Penalties in Real Applications

In real applications, memory accesses are much more complicated than what is observed with the microbenchmarks. The microbenchmarks are atomic in nature and can be sensitive to the exact details of how the data accesses are coded. In the microbenchmarks discussed the objective is for only a single architectural aspect ito be explored and it explicitly investigated with sequential scheduling. In a real application, the compiler overlays as many high latency operations in parallel as possible so that the latencies are hidden. With the added complexity of the out-of-order data returns from the L2 and L3 caches on Itanium® 2 processors, the model of using simple penalties

to approximate the contributions to the stall cycles will fail in its numerical consistency. However, it can serve as a very useful guideline to determine the relative importance of which memory access issues to address.

Measuring Memory Access Penalties

Caches in the Itanium® 2 processor can return data out-of-order so memory access stalls become more complicated as more memory requests and writes are generated by a single high-level language instruction. To understand how to interpret the access stall counters in the Itanium® 2 processor,

- start with the simplest access of a single variable and modify the size of the array to force the part of the memory subsystem that must be accessed (cache or main memory).
- analyze multiple data accesses to see how conflicting addresses can cause inefficient data delivery.

In each case, this guide provides general suggestions for alleviating memory access stalls by modifying source code and the way it interacts with data.

Memory Access Stalls due to Simple Cache Misses

Whenever a load instruction attempts to access data from a data cache array that does not contain the desired data, it encounters a cache miss. All integer loads attempt to access the L1D first. All floating point loads access the L2 first.The load request is escalated up through the cache hierarchy (L1->L2->L3 for integers, L2->L3 for floats), to memory, and ultimately to disk until the desired data is found. The compiler schedules the use of the data assuming the fastest cache system satisfies the data request. As the data request is escalated through the memory hierarchy, the latency for the ultimate delivery of the data increases. Simultaneously, the cache miss and cache reference counters for each level of the hierarchy are incremented appropriately, leaving a trail for you to investigate.

Cache misses are caused when

- the cache line containing the desired data has been replaced by other data
- the cache line has not yet been loaded with the required data

Microbenchmark for Simple Access

The microbenchmark used in this discussion for determining the memory access stall penalties for missing the highest speed caches is shown in the following example.

`do_read_inner:`
`{ .mmi`
`add r29 = r30, r33;;`
`ld4 r28 = [r31]`
`and r30 = r29, r34;;`
`}`
`do_read_branch:`
`{ .mib`
`mov r29 = r28`
`add r31 = r30, r35`
`br.cloop.dptk.few do_read_inner;;`
`}`
	`do_calibrate_inner:`
	`{ .mmi`
	`add r29 = r30, r33;;`
	`nop.m 0`
	`and r30 = r29, r34;;`
	`}`
	`do_calibrate_branch:`
	`{ .mib`
	`mov r29 = r27`
	`add r31 = r30, r35`
	`br.cloop.dptk.few do_calibrate_inner;;`
	`}`

The latency is forced on every iteration of the loop by trying to move the data from r28 to r29. The target cache for the test is forced simply by correctly setting the range within the accessed buffer. This is controlled by the contents of r34. The baseline can then be determined from the version on the right, which is identical except that there is no load executed.

The result of executing such a code is summarized in the following table.The latencies derived with this microbenchmark should not be interpretted as absolute results. They are used here to determine relations that can serve as guides in application analysis

Memory subsystem accessed	Latency (in Cycles)
L1	1
L2	5
L3	13.3
Memory	209.6

These are the measured penalties for integer data access. The L3 latency is slightly longer than the stated value in the Itanium® 2 Processor Reference Manual for Software Development and Optimization. Observing the minimum L3 latency requires a more carefully constructed test that is not quite as general. For the purposes of this guide we will use the measured number as all other tests will be based on the one shown. The latency from main memory is also a function of the

chipset. The latencies quoted in this document were measured on an Intel platform using the 870 chipset. Note that one cycle is added to the result from the microbenchmark program due to the cycle having spent issuing the second stop bit in the first bundle.

If the code is modified to load floating point data, as shown below, the penalties become slightly different due to the change in the path the data uses to arrive in the register file. Floating point data is loaded directly from the L2 Cache.

do_read_inner: { .mmi add r29 = r30, r33;; ldfs f28 = [r31] and r30 = r29, r34;; } do_read_branch: { .mib mov f29 = f28 add r31 = r30, r35 br.cloop.dptk.few do_read_inner;; }	do_calibrate_inner: { .mmi add r29 = r30, r33;; nop.m 0 and r30 = r29, r34;; } do_calibrate_branch: { .mib mov r29 = r27 add r31 = r30, r35 br.cloop.dptk.few do_calibrate_inner;; }

The resulting latencies are shown in the following table. Note that again one cycle has to be added to the measured result for the second stop bit in the first bundle.

Memory subsystem accessed	**Penalty (in Cycles)**
L2	6
L3	13.1
Memory	209.5

With this information, it is possible to construct a model to account for the memory access stalls.The model we will construct here should only be used as a guide to assist in determining the major causes of execution ineficiency. The relationships that result will always overestimate the stall cycles associated with the occurrence event counts used. Approximately, the following relations would hold if nothing else was occurring. As discussed in the next chapter, if there are

TLB misses then the relationship for the contribution of memory access stall cycles become a little more complicated. Furthermore, always remember that multiple memory accesses shield each other as they can occur simultaneously.

Presumably, the compiler has already scheduled for optimal latency and been able to completely absorb those optimal latencies in the scheduling between data loads and data use. To calculate the contribution of cache misses to the memory access stall cycles, subtract out the minimal latencies that the compiler assumed. In an application that is integer intensive, (ignore floating point cache misses at this time) you can approximate the contribution to:

```
BE_EXE_Bubble.GRALL - BE_EXE_Bubble.GRGR ~
     (L1_Read_Misses-L3_Reads.Data_Read.All)*(5-1) +
             L3_Reads.Data_Read.Hit*(13.3-1) +
     L3_Reads.Data_Read.Miss*(209.6-1)
```

In a floating point intensive application, where you could ignore integer load cache misses, the contribution to BE_EXE_Bubble.FRALL is:

```
BE_EXE_Bubble.frall~
     L3_Reads.Data_Read.Hit*(13.1-5)
     L3_Reads.Data_Read.Miss*(209.5-5)
```

where it is assumed that the compiler scheduled for the fastest appropriate cache latency. There is a difficulty with the relations just shown in that the OzQ coalesces the cache misses it escalates to L3 and the system bus. Consequently the contribution due to misses in L1 that are satisfied by L2 is overestimated and the contribution due to the recirculating coalesced misses has not yet been included. This will be discussed further in the section about multiple misses to a single cache line.

These components can then be normalized by the instructions retired to compute their approximate contribution to CPI.

For integer intensive code, compute:

```
     CPI(L2 Hits) = (L1_READ_MISSES-L3_Reads.Data_Read.All)*4/IA64IR
     CPI(L3 Hits) = (L3_Reads.Data_Read.Hit*12.3/IA64IR
     CPI(L3 Misses) = L3_Reads.Data_Read.Miss*208.6/IA64IR
```

Whereas for a floating point intensive code, compute:

```
     CPI(L3 Hits) = L3_Reads.Data_Read.Hit*8.1/IA64IR
     CPI(L3 Misses) = L3_Reads.Data_Read.Miss*204.5/IA64IR
```

The point of the above exercise is to determine the dominant contributions to inefficient execution of the program. While L1 cache misses count at a much higher rate than L3 cache misses, L3 cache misses carry a penalty which is 50 times greater than the bulk of L1 cache misses that are presumably satisfied by L2

It should be noted that we use the event L3_Reads.Data_Read.All to get the L2 data Misses. The event L2_Misses also counts the cache misses due to writes. As the caches are write through, a write to a missing cache line causes that cache line to be retrieved, thus a cache miss is generated.

Optimizing Cache Miss Stalls

If the expressions in the previous section (integer or FP, as appropriate) are significant contributions to the total CPI, you must reduce the cache misses contributing to the largest component. The most obvious solution is to check that all loops are accessing data sequentially in memory so that the cache line structure is used efficiently. In the case of multi-dimensional arrays in C, the loops should access the last index, in Fortran, the first index.

Raising the optimization level to /O3 will invoke the high level optimizer (HLO). The compiler will then generate prefetch instructions as it deems beneficial. This does not reduce the cache misses as the prefetch will still generate a cache miss condition. It will however allow the binary to request the data earlier than it would normally allowing the additional latency to the slower memory subsystem to be absorbed.

If you have already raised the compiler optimizations to the highest level then the prefetch intrinsic, in ia64intrin.h, can be used in the source code to try to fetch the data far enough in advance to hide the access latency.

void __lfetch(int lfhint, void *y)

Where the intrinsic will generate the lfetch.lfhint instruction. The value of the first argument specifies the hint type which controls which cache level the data should be brought into. For example floating point data should only be prefetched to the L2 cache.

__lfetch(nt1, address)

Assuming that the access ordering has been done correctly, “block” the data access to take advantage of the fastest cache and the entire hierarchy. The systematic approach to do this is outlined as follows:

- reduce the problem size to one that fits in the fastest cache, L1D for integers, L2 for floating point.
- create a module that copies the subset of the data into a temporary array that allows all accesses to be sequential
- tune an inner kernel to run at maximum speed

At this point, divide the larger problem into pieces of the size that can be consumed by the copy and kernel modules. The main body then becomes a loop through the appropriately-sized blocks.

It is important to realize that the copy into the temporary array not only ensures sequential access but that it has a second effect which can occasionally be even more important. This is forced cache line replacement. Consider a (worst case) 4096 by 4096 matrix of doubles which is blocked into 32 by 32 sub-matricies. The sub-matricies are stored in memory as a series of short sequential segments, the size of 2 cache lines. Their base addresses are separated by 32768 bytes (15 bits in the address), due to the Leading Dimension (4096) of the array. The associative set (row) within the L2 cache is determined by bits 8 through 15 of the address, as L2 has 256 rows of 8 cache lines 128 bytes long. The net result is that the 32 by 32 submatrix can only be stored in two of the 256 associative sets and only one quarter of the sub-matrix can be resident in the L2 cache at any given time. Due to this kind of effect, in a real problem there may be a very poor occupation of the entire cache, with the data all being forced into a small number of associative sets.

Even if there is a significant component of BE_EXE_Bubble.GRALL - BE_EXE_Bubble.GRGR or BE_EXE_Bubble.FRALL that is accounted for by the cache miss penalties shown above, consider other possible sources for memory access stalls because of the overlapped accesses the OzQ supports. The following section is a discussion of the next most likely source, Bank/Address conflicts.

Bank/Address Conflicts and Their Contribution to EXE Stall Cycles

When memory access stall cycles are a significant issue and cannot be reasonably accounted for by the approximations discussed in the last section, then investigate other sources for the stalled cycles. Microprocessor cache architectures frequently have access structures that allow for very low latencies but in some circumstances this is not applicable. You encounter one of the more common causes of restrictions when the program needs to load multiple pieces of data at the same time. This means that the compilers must generate code that issues two load instructions on the same cycle. Under these conditions, address conflicts can arise causing less than optimal data access latencies. This effect is more likely to occur in floating point intensive applications with their large data samples but is by no means restricted to them. Both cases are explicitly discussed below.

The L2 Cache is 256KB, 8-way associative with 128 byte cache lines. It is constructed of 16 banks that are 16 bytes wide. When multiple loads are issued within a cycle that must be satisfied by L2, complications can arise if there are address or bank conflicts. If the loads can be satisfied by L1 there are no conflicts and both loads occur in one cycle.

When multiple memory accesses are made to data stored in the same L2 banks, a conflict arises. This impacts the latency as different access mechanisms are invoked by the L2 OzQ. It is most apparent when floating point data is loaded as there is no additional interaction with the L1 cache.

L2 bank conflicts cause several architectural event counters to accumulate. These counters can be used to identify the issue and its severity. L2_OZQ_Cancels1.Bank_CONF is incremented for every L2 access that experiences a bank conflict. L2_Bypass is also incremented by both loads if there is a bank conflict. So identifying these is quite easy. Comparing these counts to L2_Data_References.L2_Data_Reads gives the fraction of L2 reads that are associated with bank conflicts.

The drill-down feature of the VTune analyzer enables you to determine exactly which lines of code are responsible for the conflicts. At this point, run the code in a debugger to analyze the addresses of the data being loaded. You can then reorganize the order of the floating point data accesses to avoid the bank conflict.

In a complex piece of code, the event skid as displayed by the VTune analyzer may make the exact location of the bank conflict confusing. If the Bank_Conflict subevent has skidded to an obviously incorrect place, use other events to identify the correct location of the bank conflict. The address conflicts and the associated bank conflicts require multiple loads per cycle. So look at the disassembly view of the executable intermingled with the source code, in the VTune Analyzer to help uniquely identify the location. The Data_EAR_Event can also be very useful for this. By selecting the subevent with a latency over the minimum, umask =1, latency >= 8 cycles, long latency accesses can be localized. If the event L3_READS.DATA_READ.ALL is used in conjunction, a reasonable condition would be a higher than normal ratio. That means long latency loads that do not miss in L2. Obviously all loads that miss L2, satisfy the latency condition. Remember that the EAR events are exact in their location but are inexact in regards to number. The hardware samples a fraction and that fraction can depend on how the program interacts with the cache access hardware.

Floating Point Data and L2 Bank Conflicts

Microbenchmark for FP Data Conflicts

To measure the penalty for floating point loads with bank conflicts, modify the latency microbenchmark to use two 256 byte aligned buffers. Then run the assembler shown below, within a high-level loop that progressively bumps the base address stored in r36 of the second buffer by 16 bytes. This steps the relative alignment of the accesses in multiples of the bank widths. If the buffers are both 256 byte aligned then the conflict occurs on the first iteration of the outer loop and repeats every 16 iterations of the outer (relative alignment driving) loop. As always the code on the right is used to subtract off the baseline where no loads are performed and no

memory access stalls encountered.Note that is all of the multiple access discussions that follow the latencies measured are due to the delivering the data for the second load. The data for the first load is delivered with the normal single access latency.

do_read_inner: { .mmi add r17 = r16, r33;; add r29 = r30, r33 and r16 = r17, r34;; } { .mmi ldfs f26 = [r15] ldfs f28 = [r31] and r30 = r29, r34;; } { .mfi add r15 = r16, r36 mov f27 = f28 nop.i 0 } do_read_branch: { .mfb add r31 = r30, r35 mov f25 = f26 br.cloop.dptk.few do_read_inner;; }	do_calibrate_inner: { .mmi add r17 = r16, r33;; add r29 = r30, r33 and r16 = r17, r34;; } { .mmi nop.m 0 nop.m 0 and r30 = r29, r34;; } { .mfi add r15 = r16, r36 mov f27 = f28 nop.i 0 } do_calibrate_branch: { .mfb add r31 = r30, r35 mov f25 = f26 br.cloop.dptk.few do_calibrate_inner;; }

Penalty and Correction for FP Data Conflicts

Running this benchmark from a high-level loop shows that the bank conflict results in an additional latency of 6 cycles. You can remove the additional 6 cycles of latency by shifting the base address by 16 (or 32, 48, 64…up to 240) bytes. Remember that the banking structure spans 256 bytes.

Access Mode	Latency
Single Access	6 Cycles
Double Access with no Bank Overlap	6 Cycles
Double Access with a Bank Overlap	12 Cycles

You can compute the upper limit to the contribution to CPI due to floating point loads that encounter bank conflict with the following relation. It will overestimate the true effect but can be used as a guide.

```
CPI(FP Bank Conflicts) = L2_OZQ_Cancels1.Bank_CONF*3/IA64IR
```

The penalty of 6 cycles is divided by two as L2_OZQ_Cancels1.Bank_CONF counts every load that has a conflict and thus “double counts” the number of loads canceled and re-issued. This is of course the upperbound on the penalty due to the bank conflicts. If you are dealing with a looping algorithm and have unrolled the loops (or if the compiler has done this for you) then more than the minimal latency can be absorbed by the scheduling of the instructions. Even so, removing the bank conflicts will reduce the OzQ activity and can improve the throughput of L2 access.

Loop Unrolling and Bank Conflicts

Removing bank conflicts in many cases is actually rather simple. Consider the double precision matrix multiply in Fortran again:

```
Do k=1,MAX
   Do j=1,MAX
              Do i=1,MAX
              a(i,k)=a(i,k) + b(i,j)*c(j,k)
              enddo
   enddo
enddo
```

The first thing to improve performance is to unroll the inner loop.

```
 Do k=1,MAX
   Do j=1,MAX
              Do i=1,MAX
```

```
                    a(i,k)=a(i,k) + b(i,j)*c(j,k)
                    a(I+1,k)=a(I+1,k) + b(I+1,j)*c(j,k)
                    a(I+2,k)=a(I+2,k) + b(I+2,j)*c(j,k)
                    a(I+3,k)=a(I+3,k) + b(I+3,j)*c(j,k)
                    enddo
        enddo
    enddo
```

While this coding improves the performance significantly, it may result in many bank conflicts caused by loading successive elements in memory during the same cycle. The simple solution is to interleave the unrolled lines as shown as follows:

```
    Do k=1,MAX
        Do j=1,MAX
                    Do i=1,MAX
                    a(i,k)=a(i,k) + b(i,j)*c(j,k)
                    a(I+2,k)=a(I+2,k) + b(I+2,j)*c(j,k)
                    a(I+1,k)=a(I+1,k) + b(I+1,j)*c(j,k)
                    a(I+3,k)=a(I+3,k) + b(I+3,j)*c(j,k)
                    enddo
        enddo
    enddo
```

This results in separating the loads from the same banks (adjacent addresses in memory) by at least a cycle thus removing the bank conflict. In the case of single precision data (4 bytes), a greater degree of unrolling (over j) and a more complex interleaving is required to remove the bank conflicts.

In a real measurement of the above example, interleaving improves performance by over 10%.

Integer Data and L2 Bank Conflicts

Access to cacheable integer data always goes through the L1D cache. This complicates the issue of addressing conflicts and their impact on the flow through the core pipeline. In order to understand addressing conflicts with integer data, you must comprehend the data flow from the caches in a bit more detail.

Integer address conflicts are an issue only when multiple integer data accesses require data from the L2 cache simultaneously. There are essentially no issues associated with multiple data access directly from the L1D. It has 2 load and 2 store ports. The load ports are fully dual-ported meaning

that any two load addresses can be read from the L1D in parallel without conflict. Stores access the L1D data array in 8 groups that are 8 bytes wide. Stores do have the potential for conflicts but special hardware is provided to limit these conflicts from impacting performance.

When considering the latency of multiple integer misses to dindependent cache lines, the bandwidth between the caches can impact the overall latency for the multiple data requests from L2. In addition to the requested data, entire 64 byte L1 cache lines are brought in. Thus multiple accesses (requiring multiple fills) that might suffer addressing conflicts between the L2 banks have the additional bottleneck of having to transfer multiple cache lines. In fact, multiple integer accesses with no addressing conflicts encounter additional latency as only one L1 cache line can be updated from L2 at a time.

The address conflicts are complicated by the fact that not only must the requested data not occupy the same L2 bank but that the entire cache lines associated with the data not overlap. The net result is that for integer data address conflicts, the L2 cache behaves like it has 4 banks which are 64 bytes in width. As the data paths are different from those for floating point data (which load directly to the floating point register file from L2), the penalties are also different.

Microbenchmark for Integer Bank Conflicts

To study the address conflicts of integers loaded from L2, modify the floating point access microbenchmark to use integer loads and move instructions as shown below:

<table>
<tr>
<td>

```
{ .mmi
add r17 = r16, r33;;
add r29 = r30, r33
and r16 = r17, r34;;
}
{ .mmi
ld4 r26 = [r15]
ld4 r28 = [r31]
and r30 = r29, r34;;
}
{ .mmi
add r15 = r16, r36
mov r27 = r28
nop.i 0
}
do_read_branch:
{ .mib
add r31 = r30, r35
mov r25 = r26
br.cloop.dptk.few
do_read_inner;;
```

</td>
<td>

```
{ .mmi
add r17 = r16, r33;;
add r29 = r30, r33
and r16 = r17, r34;;
}
{ .mmi
nop.m 0
nop.m 0
and r30 = r29, r34;;
}
{ .mmi
add r15 = r16, r36
mov r27 = r28
nop.i 0
}
do_calibrate_branch:
{ .mib
add r31 = r30, r35
mov r25 = r26
br.cloop.dptk.few
do_calibrate_inner;;
```

</td>
</tr>
</table>

Again, you can remove the address conflicts by adjusting the base address of the second buffer stored in r36. Do this from a main program that calls the assembler function from within a loop which adjusts the second buffer's base address in steps of 16 bytes (the L2 bank width).

Penalties for Integer Bank Conflicts

The pattern for integer data conflicts is different than that for floating point data. If the integer addresses overlap such that their cache lines lie in overlapping L2 banks, then there is a conflict. This translates into a one in four possibility instead of a one in 16 for a floating point conflict.

On running the microbenchmark, you will find that if the addresses overlap within the 64 byte window that corresponds to an L1 cache line, the latency (which is due to the second load) is increased to 11 cycles. If the two addresses do not overlap, within the definition being discussed, then the latency is 7 cycles. Compare these numbers to the 5 cycle integer latency for a single access from L2 and the 1 cycle access from L1 that the compiler uses for scheduling.

Access mode	Latency
Single access from L1	1 cycle
Single access from L2	5 cycles
Double access from L2, no overlap	7 cycles
Double overlapping access from L2	11 cycles

Integer-Floating Point Data Bank Conflicts

While address conflicts can in principle occur between integer and floating point data, in reality this is extremely rare. This is because the floating point unit only uses floating point registers and the integer ALU's only use general registers. So these conflicts can occur only in situations where integer and floating point are intermingled in calculations performed within bundles. Calculations that use both integers and floating point data together in a single expression result in data conversion (hardware recasting of data) in order to use the two data types together. So the compilers probably do not schedule the loads within the same cycle. Keep in mind that if loads arrive at the OzQ even one cycle apart there is no bank conflict.

Microbenchmark for Integer-FP Data Bank Conflicts

You can easily modify the microbenchmark to issue a floating point and integer load together, followed by an integer and floating point move on the next cycle. As always with these tests the measured latency increase is due to the second load as the first load is completed normally. The resulting latency table can then be determined as shown below:

Access Mode	Latency
Single Integer Load from L2	5 Cycles
Single Floating point load (from L2)	6 Cycles
Integer and FP Load with no Bank Overlap	7 Cycles (for both to be available)
Integer and FP Load with a Bank Overlap	11 Cycles

You can see that the penalties are the same as for the integer conflicts and the contributions to CPI are the same as shown on the previous table. Further as a 64 byte cache line is brought into L1, the L2 cache acts as if there were only 4 banks that were 64 bytes wide.

Penalties and Corrections for Integer Accesses from L2 with Bank Conflicts

You can approximate the contribution to CPI due to integer accesses to L2 with bank conflicts for integer-intensive code with the following computation where we assume that floating point access bank conflicts can be ignored as follows. As always the relation should only be used as a guide.

```
CPI(L2 Integer Bank Conflicts)=
                    L2_OZQ_CANCELS1.BANK_CONF*5/IA64IR
```

Again, the penalties for bank conflicts are divided by two to avoid double counting. You must divide the L2_OZQ_Cancels1.Bank_Conflict by two to correctly count the bank conflicts.

As with the case of floating point bank conflicts in a looping code where the bank conflict is between two different buffers, the solution to the bank conflict issue with integers (4 cycle penalty) is to add some padding to the base address of the second buffer. As before, start with a 256 byte aligned buffer address alignment declaration of the form,

```
Buf = (UINT64) malloc(sizeof(mybuffer));
buf = buf + 256 - ((UINT64)buf%256);
```

and then adjust one of the buffers by moving it enough so that it moves into the next 64 byte piece of the L2 structure.

The more common problem however, is due to loading integers that are stored adjacently. In this case, the best solution is to try to overcome the L1D cache miss rather than work on the access conflict. If you truly cannot overcome the L1D cache miss and this is a dominant source of stall cycles, then you have two options:

- Separate the first two references to the adjacent data so you can use one load to bring the data into the L1D cache.
- Get the two pieces of data into different cache lines. This requires restructuring your data storage, for example by using multiple structures.

OzQ Recirculations and Multiple L2 Cache Misses

Multiple simultaneous attempts to access a cache line not resident in L2 (nor L1D) will result in the OzQ recirculating the secondary accesses. The first access causes an L2 cache miss and an escalation to L3 (and the system bus, if required) to recover the data. The secondary accesses go into a recirculating state until the cache line is updated in L2.

To determine the impact on the observed latency, the bank conflict microbenchmark is modified so that the two base addresses point to the same buffer but with a fixed relative offset. The offset is set in the driving program in a loop to also explore any additional depedencies on the relative bank accesses. The handling of secondary misses is a complex process in the Itanium® 2 processor cache structure . Consequently, the numbers determined with the microbenchmarks may differ significantly from what would be observed in a real application and represent the most uncertain extrapolations of the work presented in this text. As with all the results presented in this text, what follows should only be used as a guide to estimate if secondary misses are creating a significant execution inefficiency in your application.

The total latency determined from this microbenchmark is summarized as follows:

- A secondary integer cache line miss in L2 that can be satisfied by L3 has a 25 cycle latency for both deliveries to be satisfied
- A secondary floating point cache line miss in L2 that can be satisfied by L3 has a 28 cycle latency for both deliveries to be satisfied if they access the different banks
- A secondary floating point cache line miss in L2 that can be satisfied by L3 has a 26 cycle latency for both deliveries to be satisfied if they access the same bank
- A secondary integer or floating point cache line miss in L2 that is satisfied by main memory has a 232 cycle latency for both deliveries to be satisfied

These latencies were measured with a microbenchmark where both loads were issued on the same cycle. We can determine a penalty for secondary miss recirculations using these latencies which can be used to estimate if this contributes significantly to the execution inefficiency. As the L3_READS.DATA_READ events get updated for the primary miss the penalty for the cache miss will have already been accounted for by the expressions determined earlier in this chapter. Only the incremental penalty needs to be estimated.

The incremental latencies are are between 11 and 14 cycles when the data can be updated from L3 and 22 cycles when it must be incremented from main memory. Therefore an average value of 17 cycles is probably adequate for estimating the incremental penalty due to secondary misses. This should only be used as a guide in your analysis as should not be interpreted in any other way.

The relevant occurrence events for this are the L2_FORCE_RECIRC and its subevents .

- L2_FORCE_RECIRC.SAME_INDEX measures the number of times a recirculation was associated with multiple accesses to a single associative set on a single cycle. It can increment by more than one count per cycle.
- L2_FORCE_RECIRC.L1W measures the number of recirculations initiated by a reference to a cache line that was missed on the previous cycle
- L2_FORCE_RECIRC.OZQ_MISS measures the number of recirculations initiated by a reference to a cache line that was missed between 2 and 5 cycles earlier

- L2_FORCE_RECIRC.FILL_HIT measures the number of recirculations initiated by a reference to a cache line that was missed more than 3 cycles earlier, that has not yet been updated.

To count all the secondary miss recirculations, all four of these events must be used. Unfortunately that does have the risk of overestimating the number of secondary cache line miss recirculations due to double counting and the definition of the SAME_INDEX subevent. The OZQ_MISS and FILL_HIT subevents both increment when the delay between the references is between 3 and 5 cycles. The SAME_INDEX subevent can increment when there are multiple references to a cache line (on the same cycle) that is already due for updating, thereby double counting with one of the other three subevents. It will also increment if there are multiple cache misses to different cache lines within the same associative set.

Further it cannot be absolutely determined if multiple miss recirculations are associated with L3 hits or misses. Consequently an average value of the 17 cycles can be used to get an estimate of the incremental penalty for secondary misses. This leads to a relation for the contribution to BE_EXE_BUBBLE due to secondary misses in L2 of

```
Contribution to BE_EXE_BUBBLE = (L2_Force_RECIRC.Same_Index +
                               L2_Force_RECIRC.L1W +
                               L2_Force_RECIRC.OZQ_MISS +
                               L2_Force_RECIRC.FILL_HIT)*17
```

and the corresponding contribution to CPI would be determined by dividing that by IA64IR.

The most obvious resolutions to the incremental latencies for secondary misses were stated earlier in this chapter. Certainly the best is to avoid the cache miss latency entirely by either

- building the program with /O3 optimization to generate the prefetches
- using prefetch intrinsics

Alternatively, the secondary miss might be avoided by

- sepperating the source lines that access the same structure or array sufficiently to allow the cache line to update from the first access. This will take about 10 cycles if the data is in L3. Note: this approach really isn't possible if the data is in main memory.
- Alternatively replacing arrays (or linked lists) of structures with structures of arrays will avoid putting the elements in the same cache line.

BE_L1D_FPU_Bubble for Stalls due to L1D and FPU Micropipelines

The BE_L1D_FPU_Bubble event accumulates stall cycles caused by the micropipelines associated with the L1D and FPU stalling the core pipeline at the DET stage.

As always, first determine the dominant source of the performance bottlenecks contributing to this event. Then apply the appropriate optimizations to remove those execution inefficiencies. The stall cycles accumulated by this counter are dominated by memory access stalls that have a different architectural basis than those accumulated by the BE_EXE_Bubble event. The stalls accumulated in this event have to do with blockages in the transfer of data and not just longer latencies than the compiled code could absorb. Consequently, you may need a different approach to remove these stalls.

Hierarchical Structure of BE_L1D_FPU_Bubble Subevents

The table below shows the subevents of BE_L1D_FPU_Bubble. The most common contributions are the Data Cache Unit (DCU) Recirculating (subevent L1D_DCURECIR).

The table is organized as a hierarchy.

- The total, ALL is divided into L1D and FPU components.
- The L1D component is divided into a large number of subevents of which five dominate the contributions encountered by applications.

Keep in mind that subevents are not prioritized and that a given cycle can be double-counted between subevents, but the sums working down the tree structure are usually close

Table 7-14 BE_L1D_FPU_Bubble subevents

Extension	PMC.umask [19:16]	Description
ALL	b0000	Back-end was stalled by L1D or FPU
FPU	b0001	Back-end was stalled by FPU.

Table 7-14 BE_L1D_FPU_Bubble subevents

Extension	PMC.umask [19:16]	Description
L1D	b0010	Back-end was stalled by L1D. This includes all stalls caused by the L1 pipeline (created in the L1D stage of the L1 pipeline which corresponds to the DET stage of the main pipe).
L1D_FULLSTBUF	b0011	Back-end was stalled by L1D due to store buffer being full
L1D_DCURECIR	b0100	Back-end was stalled by L1D due to DCU recirculating
L1D_HPW	b0101	Back-end was stalled by L1D due to Hardware Page Walker
---	b0110	(* count is undefined *)
L1D_FILLCONF	b0111	Back-end was stalled by L1D due a store in conflict with a returning fill.
L1D_DCS	b1000	Back-end was stalled by L1D due to DCS requiring a stall
L1D_L2BPRESS	b1001	Back-end was stalled by L1D due to L2 Back Pressure
L1D_TLB	b1010	Back-end was stalled by L1D due to L2DTLB to L1DTLB transfer
L1D_LDCONF	b1011	Back-end was stalled by L1D due to architectural ordering conflict
L1D_LDCHK	b1100	Back-end was stalled by L1D due to architectural ordering conflict.
L1D_NAT	b1101	Back-end was stalled by L1D due to L1D data return needing recirculated NaT generation.
L1D_STBUFRECIR	b1110	Back-end was stalled by L1D due to store buffer cancel needing recirculate.
L1D_NATCONF	b1111	Back- end was stalled by L1D due to ld8.fill conflict with st8.spill not written to unat.

Note that many of the subevents of BE_L1D_FPU_Bubble usually have negligible contributions to the stall cycles for compiled code. This is because they deal with access conflicts associated with NAT bits, application registers, control registers and load.acq/st.rel memory fencing instructions. These have not been seen to contribute significantly for compiled high-level language

code. The L1D_DCS, L1D_LDCONF, L1D_LDCHK, L1D_NAT and L1D_NATCONF subevents fall into these categories and will not be discussed further. As compiler technology advances more aggressive use of speculation may result in these events accumulating significant values.

If you notice that these events contribute significantly to stall cycles, contact your Intel representative or use the https://premier.intel.com support web page to get assistance.

L1D and FPU - Dominant Components of BE_L1D_FPU_Bubble

The L1D and FPU components of the BE_L1D_FPU_Bubble counter form the primary division of the stall cycles that separate out each of the micropipelines. Remember that subevents are not prioritized so you cannot apply an exact sum rule. An approximation to a sum rule, however, is in general sufficient to isolate and understand the dominant bottlenecks.

L1D components are far more likely than FPU to contribute substantially to the stalled cycles and expose coding issues that you can easily correct.They can frequently have a considerable contribution to CPI.

The FPU subevents, rarely accumulate any counts in applications and the components of the SPEC benchmarks that have been studied. So, this chapter will mostly discuss the components of L1D but just for completeness of information, FPU will be briefly discussed first.

FPU

The FPU subevent counts cycles for cases where the FPU micropipeline stalled to ensure correct behavior. There are two cases that cause such stalls. The common one is a chain of floating point instructions where the output of the first is used as input to the second:

```
Fma f33=f34,f35,f36;;
Fma f37=f38,f39,f33
```

The use of f33 as input to the second fma instruction causes the second instruction to stall until the first has completed returning data.This is' scoreboarding' the instructions. The output registers are tagged as invalid until the data is finished being written to them. In this case, the stalled cycles are accumulated in the EXE stage of the core pipeline in the BE_EXE_Bubble.FRALL counter.

The less common occurrence is due to the need to detect exceptions at the DET stage of the pipeline. Floating point exceptions can take several cycles to occur in which time more issue groups could be retired. In order to insure the integrity of the processor state the Safe Instruction Recognition (SIR) hardware inspects the floating point data and makes a fast determination if the floating point operation is going to safely complete. If this determination is established the core pipeline continues without interuption. If the SIR hardware determines that there is a chance of an exception occuring then the pipeline is stalled. In this case, the cycles are accumulated in the FPU micropipeline stall counter, Be_L1D_FPU_Bubble.FPU.

If this subevent has significant counts the you should look at the events FP_TRUE_SIRSTALL and FP_FALSE_SIRSTALL. These will confirm that the SIR is stalling the pipeline.

To remove these stalls the calculation needs to be regularized to avoid denormalization of the floating point results. This can be done by shifting single precision to double precision, recompiling with flush to zero option enabled or reformulating the algorithm to avoid extremely small or large numbers.

L1D

The number of stalls accumulated in the L1D subevent is mostly divided between the L1D_DCURECIR, L1D_L2BPRESS, L1D_TLB, L1D_HPW, L1D_STBUFRECIR and L1D_FULLSTBUFF. Programs usually do fewer writes than reads, and rarely do so many as to fill the store buffers. The more usual problems occur with load access of data and the conflicts associated with those loads. This discussion will cover the simpler component, L1D_FILLSTBUFF first.

The L1D_FULLSTBUFF counter is fairly self-explanatory. Too many integer stores issued within a small number of cycles can fill the L1D store buffer. This in turn causes the core pipeline to stall until more store instructions can be handled. You can solve this simply by interchanging source lines so that the writes to memory are not bunched together. This is a good coding practice on any microarchitecture.

Contributions to L1D due to Data Access

The most common components of the L1D subevent that accumulate significant counts are L1D_DCURECIR, L1D_TLB, L1D_HPW, L1D_L2BPRESS, and L1D_STBUFRECIR. There are three main causes to stalling the L1D micropipeline which will be discussed in this section, DTLB misses, the L2 OzQ being full and load store scheduling conflicts. To explain the relationships between these counters a bit of detail on the L1D micropipeline will be required.

The L1 Data cache does not queue requests like the L2 OzQ does. Consequently when a request cannot be serviced it must be recirculated until it can. Core pipeline stalls to allow the L1D micropipeline to synchronize while the requested are being recirculated are accumulated in the L1D_DCURECIR (Data Cache Unit Recirculate) subevent. What needs to be determined is the cause of the DCU recirculations and the L1D micropipeline stalls.This is revealed as related counts in the other L1D subevents.

Access to integer data is tightly coupled to the L1 data cache and the two-level translation lookahead buffer. The interaction of the L1 data cache and theL1DTLB will be addressed first. The interaction of the two TLBs will be discussed afterwards.

The discussion of L1D_TLB and L1D_HPW requires familiarity with the architecture of the virtual page access system. This is described at the end of chapter 3.

Measuring the Impact of DTLB Misses

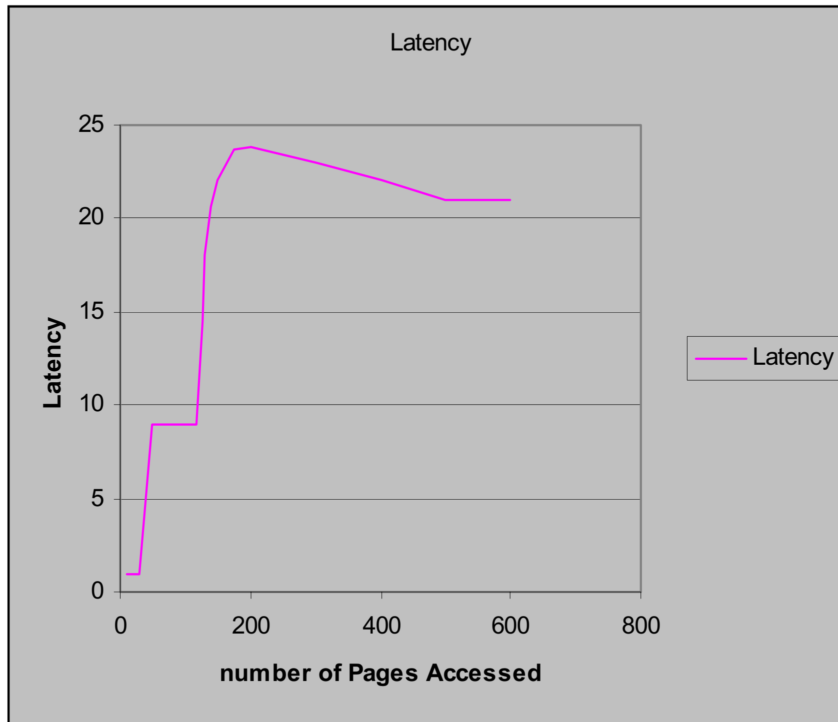
You can easily measure the effect of DTLB misses and invoking the HPW by modifying the microbenchmark used to determine cache access latencies. The stride of the address increments is merely extended to be the virtual page size (plus one cache line, to use all the associative sets), forcing a new page to be accessed each time. If more than 32 pages are accessed, then the L1DTLB misses. This in turn causes an L1 cache miss because an L1 hit requires a valid L1DTLB entry.

If the required entry is stored in the larger L2DTLB, all that is required is to transfer the entry to the L1DTLB and update the cache line from L2. This transfer increments the TLB counter L1DTLB_TRANSFER by one. In this case, the total stall is 9 cycles. Some of these stall cycles are accumulated in the L1D (4), split between L1D_DCURECIR (3) and the L1D_TLB (1) subevents. The other 5 are added in BE_EXE_Bubble.GRALL as there is also a required access to L2 that is required to update the cache line in L1.

As the number of pages accessed increases towards 128, the L2DTLB starts missing as well. This increments the L2DTLB_MISSES event counter. At this point, the Hardware Page Walker (HPW) is invoked to locate the data. The HPW uses one entry in the L2DTLB for each page of entries of Virtual Hash Page Table (VHPT) it accesses. Most operating systems use the short 8 byte format for the VHPT so there are 1024 etries on a standard 8 KB page. Further the OS can reserve up to half the entries though in practice only a few (>5) are reserved. This means that less than 128 pages can be accessed using only the L2DTLB.

If you run the microbenchmark accessing 200 pages with a stride of one page (plus one cache line), the access latency becomes 23 cycles. Four cycles are assigned to BE_EXE_Bubble.GRALL. The rest are accumulated in the BE_L1D_FPU_Bubble.L1D stall cycle counter. As we proceed down the L1D chain, almost all of the cycles are accumulated under the L1D_DCURECIR and the L1D_HPW subevents. Under these circumstances, the counters L2DTLB_Misses, DTLB_Inserts_HWP and DTLB_Inserts_HWP_Retired are all incremented with each miss.

The following graph shows the integer load access latency as a function of the number of pages accessed. Notice the step where the L2DTLB is updating the L1DTLB and then the plateau where the HPW is handling the virtual to physical translations.

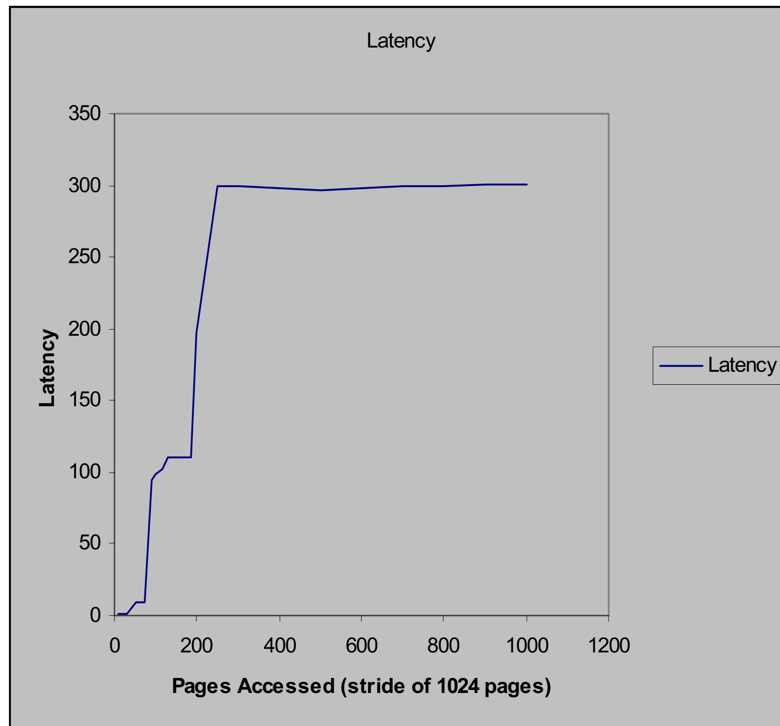


When you increase the stride of the accesses from one page (plus one cache line) to 1024 pages (plus one cache line), the access latency changes quite dramatically. Again, you have the plateau when the L2DTLB can transfer the required information to the L1DTLB, and the L2 cache can update the cache line as before, causing a latency of 9 cycles. Above 75 pages, the latency starts to rise as the HPW can no longer find the location of the VHPT data in the L2DTLB.

When the virtual to physical translations of the pages of VHPT data can no longer be found in the L2DTLB a VHPT fault is raised. This must be handled by the OS and the resulting latency is OS dependendent. The handlers are typically very fast. The data shown below was measured and can be considered as an example.

The exception handler exhibits two modes. The first plateau appears at around 100 to 200 pages and indicates a data delivery latency of around 100 cycles. At this point, a measurable fraction of the latency is involved in handling exceptions and dealing with front end stalls (17% and 10% respectively). The balance is split between BE_EXE_Bubble.GRALL and BE_L1D_FPU_Bubble, with the bulk of the latter being due to the L1D_HPW subevent.In this case the L1D_HPW and L1D_DCURECIR subevents are about equal to each other and equal to the overall L1D sum, showing that two subevents count the same cycles.

Finally, as the number of pages increases above 200, you will find that the number of L3 misses increases dramatically causing the latency to grow to 300 cycles. Most of the increase is accumulated in BE_EXE_Bubble.GRALL (76% of the total) with each of the other 3 pieces of the total (L1D, FLUSH, and FE) doubling over what they were in the exception handlers faster mode.



In all of the above cases, decrease the amount of memory that gets churned through to improve the execution efficiency. It may also help to block the data accesses used by calculations to stay more locally focussed. In the cases where the L2DTLB starts missing, there is another option of increasing the page size. The L1DTLB only supports a fixed 4KB page and uses multiple entries to support larger system page sizes. The L2DTLB and the HPW/VHPT can support substantially larger pages. So increase the page size to allow these components to span larger expanses of data.This can be done with OS APIs for Windows* .NET or by rebuilding the kernel in the case of Linux.

The L1D_L2BPRESS Subevent

This subcomponent of the L1D tree is perhaps the most complicated. When the L2 access queue (OzQ) gets full, it feeds a “back pressure” to the L1D micropipeline to stop any further access requests from propagating to the L2 OZQ. This will stall the L1D micropipeline until there are free positions in the OzQ to accept more access requests This in turn stalls the core pipeline and accumulates stall cycles in this counter. To remove this type of execution inefficiency from your application you must reduce the long lifetime OzQ entires of the program’s L2 accesses. The following list gives the most likely things to consider:

- making sure that the appropriate prefetch instructions are issued to avoid cache misses
- explicitly unrolling loops and interchanging lines to remove bank conflicts.
- Reduce OzQ recirculations due to secondary misses in the L2 Cache
- use the nt1 hint on prefetch instructions/intrinsics to reduce their lifetimes in the OzQ

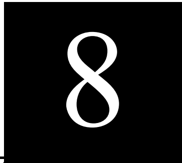
Note: you should always use the nt1 hint for prefetching floating point data and transient data where use of L1D is inappropriate.

Use the software pipelining reports and HLO reports from the compiler to gain insight into where the compiler is or is not automatically doing these things and do them explicitly. This frequently helps.

Contributions to L1D_STBUFRECIR

Cycles accumulate in this counter when there are store-load conflicts. This happens when a load follows a store within 3 cycles and both reference the same address or cache line. This should be relatively rare and in any event you can usually address it by shuffling source lines around.

BE_Flush_Bubble for Stalls due to Pipeline Flushes



The BE_Flush_Bubble counter accumulates cycles lost to pipeline flushes. These stalled cycles comprise of:

- branch mispredicts
- exceptions

You can find out directly if a stall was a branch mispredict or an exception by using the subevents BE_Flush_Bubble.xpn or BE_Flush_Bubble.bru respectively.

The following table lists all the subevents for the BE_Flush_Bubble counter.

Table 8-15 BE_Flush_Bubble subevents

Extension	PMC.umask	Description
ALL	bxx00	Back-end was stalled due to either an exception/interruption or branch misprediction flush
BRU	bxx01	Back-end was stalled due to a branch misprediction flush
XPN	bxx10	Back-end was stalled due to an exception/interruption flush
---	bxx11	(* nothing will be counted *)

In general, the most effective way to deal with significant loss of throughput due to pipeline flushes is to compile using profile guided feedback and interprocedural inlining. See Chapter 9 for an explanation of these concepts. To investigate the cause of lost cycles of this type, use occurrence events to show the details of the pipeline flushes.

Stalls due to Branch Mispredictions

You can investigate branch misprediction details using the BR_MISPRED_Detail counter. The parent counter accumulates on all branches and not just those that were mispredicted. For precise data, calculate the difference of BR_MISPRED_Detail-BR_MISPRED_Detail2 on a subevent by subevent basis:

```
( BR_MISPRED_Detail(umask=xxyy)-BR_MISPRED_Detail2(umask=xxyy) )
```

Frequently, this degree of precision is not required and the difference can be ignored. In that case, compute the total number of mispredicted branches as follows:

```
BR_MISPRED_Detail(umask=0)-BR_MISPRED_Detail(umask=1)
```

where the value with umask = 0 is the total, and the value with umask = 1 is the number of correctly predicted branches. The precise number of mispredicted branches requires the BR_MISPRED_Detail2 correction.

```
(BR_MISPRED_Detail(0)-BR_MISPRED_Detail2(0))-
(BR_MISPRED_Detail(1)-BR_MISPRED_Detail2(1)
```

There are a large number of subevents for branch mispredictions. These allow the further break down into incorrectly predicted branch direction and target. See the following table.

Table 8-16 Branch Misprediction subevents

Extension	PMC.umask	Description
ALL.ALL_PRED	b0000	All branch types regardless of prediction result
ALL.CORRECT_PRED	b0001	All branch types, correctly predicted branches (outcome and target)
ALL.WRONG_PATH	b0010	All branch types, mispredicted branches due to wrong branch direction
ALL.WRONG_TARGET	b0011	All branch types, mispredicted branches due to wrong target for taken branches
IPREL.ALL_PRED	b0100	Only IP relative branches, regardless of prediction result
IPREL.CORRECT_PRED	b0101	Only IP relative branches, correctly predicted branches (outcome and target)

Table 8-16 Branch Misprediction subevents

Extension	PMC.umask	Description
IPREL.WRONG_PATH	b0110	Only IP relative branches, mispredicted branches due to wrong branch direction
IPREL.WRONG_TARGET	b0111	Only IP relative branches, mispredicted branches due to wrong target for taken branches
RETURN.ALL_PRED	b1000	Only return type branches, regardless of prediction result
RETURN.CORRECT_PRED	b1001	Only return type branches, correctly predicted branches (outcome and target)
RETURN.WRONG_PATH	b1010	Only return type branches, mispredicted branches due to wrong branch direction
RETURN.WRONG_TARGET	b1011	Only return type branches, mispredicted branches due to wrong target for taken branches
NTRETIND.ALL_PRED	b1100	Only non-return indirect branches, regardless of prediction result
NTRETIND.CORRECT_PRED	b1101	Only non-return indirect branches, correctly predicted branches (outcome and target)
NTRETIND.WRONG_PATH	b1110	Only non-return indirect branches, mispredicted branches due to wrong branch direction
NTRETIND.WRONG_TARGET	b1111	Only non-return indirect branches, mispredicted branches due to wrong target for taken branches

Microbenchmarks for Branch Mispredictions

You can determine the penalty for the misprediction of a conditional branch with a simple loop as shown in the following code.

```
b1_2:
 {   .mfi
nop.m 0     nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0     nop.f 0
xor r11 = 1, r10;;
 }
 {   .mfi
nop.m 0
nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0
nop.f 0
cmp4.eq.unc
p7,p8=r11,r0;;
 }
 {   .mfi
nop.m 0     nop.f 0
nop.i 0
 }
 {   .mib
nop.m 0
nop.i 0
 (p8) br.cond.sptk.clr
.b2_2
 }
```

```
.b1_2:
 {   .mfi
nop.m 0     nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0     nop.f 0
xor r11 = 1, r10;;
 }
 {   .mfi
nop.m 0
nop.f 0
nop.i 0
 }
 {   .mfi
nop.m 0
nop.f 0
cmp4.eq.unc
p7,p8=r11,r0;;
 }
 {   .mfi
nop.m 0     nop.f 0
nop.i 0
 }
 {   .mib
nop.m 0
nop.i 0
 (p8) br.cond.sptk.clr
.b2_2
 }
```

`.b2_2:`
`{ .mfi`
`nop.m 0`
`nop.f 0`
`mov r10 = r11`
`}`
`{ .mfb`
`nop.m 0`
`nop.f 0`
`br.ctop.sptk .b1_2 ;;`
`}`
	`.b2_2:`
	`{ .mfi`
	`nop.m 0`
	`nop.f 0`
	`nop.i 0`
	`}`
	`{ .mfb`
	`nop.m 0`
	`nop.f 0`
	`br.ctop.sptk .b1_2 ;;`
	`}`

The register, r10, is initialized to zero before the loop. The code on the left mispredicts the branch on every other iteration of the loop. The code on the right determines the baseline as there were no branch mispredictions. The mov r10=r11 instruction is replaced by a nop.i to keep the predicate p8 from toggling values. If the branch is not taken, there is a misprediction and the pipeline must be flushed even though there is no difference in the paths. For a 1000 iterations of the loop, the code on the left takes 7000 cycles and the code on the right takes 4000. As half the loop iterations have a branch misprediction, you can conclude that the predicated conditional branch misprediction will cost 6 cycles. Note that such a simple loop incurs no larger penalties due to locating new instructions for the L1-I cache that a more realistic branch misprediction would suffer.

On Itanium® 2 processors, the branch prediction hardware is always used, the sptk branch hint merely initializes the prediction history table (PHT). This is different from Itanium® processors where such a branch hint precludes the use of the branch prediction hardware. By using the clr hint, the table is reinitialized each time.

Clearly a compiler never generates such "bad" code.

BE_RSE_Bubble for Stalls due to the Register Stack Engine

9

There are 96 general registers used for the register stacks. A deep call stack or a call stack through functions with heavy register needs can exceed this resource. Such situations require the Register Stack Engine (RSE) to spill the values stored in these registers for higher levels of a call chain to a backing store. The RSE then recovers the values as the call stack is unwound.This occurs automatically as the need arises. When the RSE is invoked, the core pipeline stalls.The BE_RSE_Bubble event accumulates these stall cycles. The RSE pushes out only the general registers to the backing store. FP registers are not swapped out to the backing store in this manner. FP registers must be spilled explicitly by the generated code. This is a rare occurrence since large numbers of FP registers are required only for local usage as in the execution of heavily unrolled pipelined loops. However, the general registers serve as the basis of the call return mechanism of argument passing and as such must be recoverable to unwind a call stack chain.

The BE_RSE_Bubble counter has a number of subevents. The complete list of subevents for this counter are listed in the following table.. However, the fact that the RSE was invoked can in itself, tell you what you need to know.

Table 9-17 BE_RSE_Bubble subevents

Extension	PMC.umask	Description
ALL	bx000	Back-end was stalled by RSE
BANK_SWITCH	bx001	Back-end was stalled by RSE due to bank switching
AR_DEP	bx010	Back-end was stalled by RSE due to AR dependencies
OVERFLOW	bx011	Back-end was stalled by RSE due to need to spill
UNDERFLOW	bx100	Back-end was stalled by RSE due to need to fill
LOADRS	bx101	Back-end was stalled by RSE due to loadrs calculations
---	bx110-bx111	(* nothing will be counted *)

If BE_RSE_Bubble is a significant source of stall cycles contributing to the primary sum rule, then you can easily determine the appropriate reprogramming actions to take in order to remove the stalls.

The RSE is invoked when there is an excessive use of general registers. Excessive use of general registers may occur in a few routines or in a heavily-used kernel routine with complex recursive algorithms. A recursive chain that requires many registers at each level quickly saturates the physical limitations of the 96-deep register file.

You can generate an example of how to invoke the RSE by writing a simple recursive algorithm to calculate the sum of

```
1/2**n

double recursive(double x)
{
     double temp, epsilon=0.001
     temp=x/2.
     if(temp < epsilon) return 0.0
     return temp+recursive(temp)
}
```

The above function calls itself recursively 10 times. If you compile the function with the /Fa (windows) option, it will yield an assembler listing that you can edit. Modify the alloc statement from:

```
alloc r33=ar.pfs,1,2,1,0
```

to one that allocates 66 local registers at each level:

```
alloc r33=ar.pfs,1,65,1,0
```

You can verify using the BE_RSE_Bubble.All counter that the RSE is invoked at each recursive level. It consumes approximately one-third of the total cycles.

Removing RSE Activity

It must be emphasized that heavy usage of general registers is the cause for invoking the RSE. If BE_RSE_Bubble contributes significantly to CPU_Cycles, then do one of the following:

- simplify the algorithms so they do not use so many general registers or
- change the compiler options

As indicated in the example (even though it was very artificial), using recursive alrgorithms in CPU-intensive parts of the program can result in very deep call stacks resulting in the need to free up registers. Recursion should be avoided in kernels on any architecture as winding and unwinding deep call stacks is never very efficient.

Another technique that might be applied to reduce RSE activity is to replace complex calculations with precomputed lookup tables and interpolation.

Another reason for RSE to be invoked could be the use of a large number of arrays and or variables, with each requiring its own unique address. In this case, tie the variables or arrays together to require fewer addresses. This may solve the problem. Alternatively, break up large complex functions into smaller simpler ones called as leaf modules may reduce the register requirements.

Using the IPO (interprocedural inlining) flag may help or aggravate the problem. You really need to check on a case by case basis. By inlining modules, there may be less duplication. On the other hand, you may complicate the calling function to the point that even more registers are required at each level of a call stack. You must simply experiment to determine the best solution. In general, interprocedural inlining should help, but there will definitely be cases where it may cause problems. Remember that the IPO flag (/Qipo for windows) really should be used in conjunction with profile guided feedback (/Qprof_gen, /Qprof_use). They work much better together.

You can also use the strategy of lowering optimization levels in non-performance critical routines. High-level optimization invokes software pipelining and loop unrolling. These two optimizations require many general registers for holding addresses. If a module is very high in the call stack and uses very little of the CPU cycles, it might reduce the register pressure to compile it at a lower optimization (/O2, /O1) to ensure that the aggressive (and high register usage) optimizations do not occur.

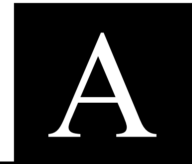
Back_End_Bubble.FE for Stalls due to the Pipeline Front End

10

Stall cycles occur when the pipeline front end is unable to supply the back end with new instructions to keep the execution going. Use the Back_End_Bubble.FE counter to accumulate stalls of this nature. These stalls are usually associated with the layout of the binary file. When the layout of the binary file does not have the active code blocks grouped together well, the instruction cache is not utilized efficiently. These kinds of stalls may or may not be associated with branch mispredictions and exceptions. On Itanium® 2 processors, the branch prediction hardware is always utilized, the branch hints merely initializing the tables. This is different than on Itanium® processors where branch hints can be used statically. Consequently, fewer severe branch mispredictions occur on Itanium 2 processors.

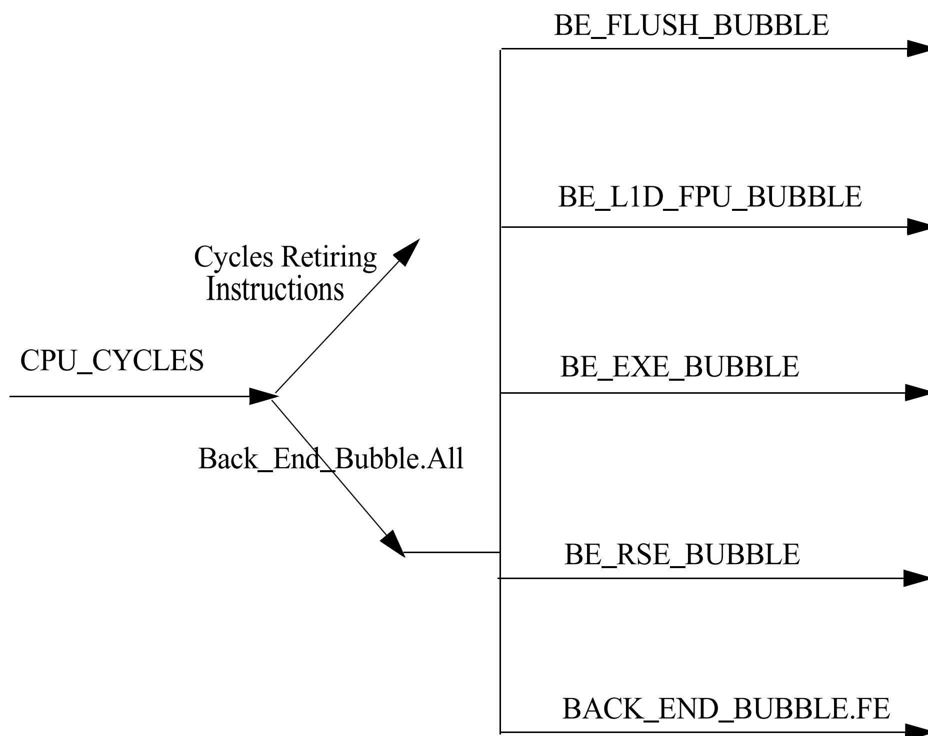
There is not much point in modifying the source to resolve this issue. In principle, it helps to make the default fall through of 'else if' blocks the dominant flow. However, simply rebuilding the application with profile guided feedback (/Qprof_gen and /Qprof_use) and interprocedural inlining (/Qipo) will accomplish more and require less work.

Appendix



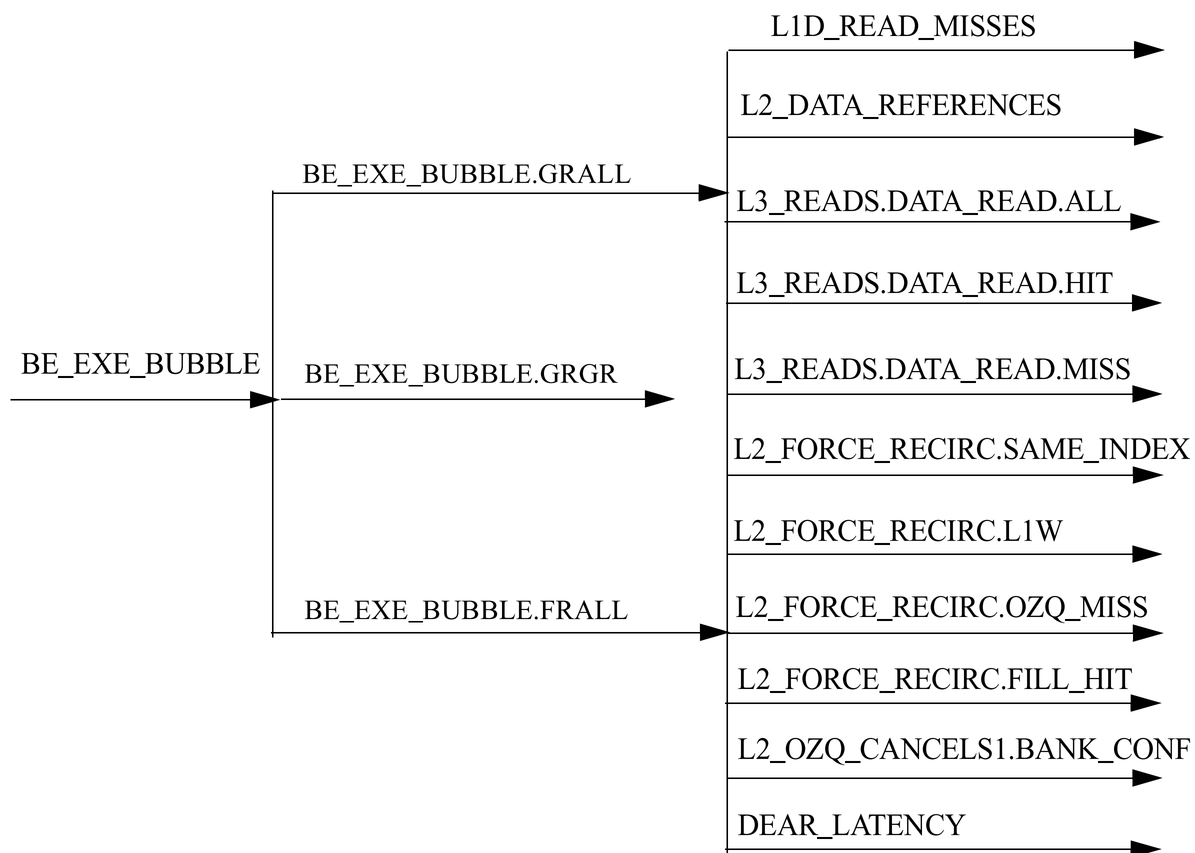
Event groups

Basic Cycle Accounting



```
CPU_CYCLES
IA64_INST_RETIRED
BACK_END_BUBBLE.ALL
BACK_END_BUBBLE.FE
BE_FLUSH_BUBBLE
BE_EXE_BUBBLE
BE_L1D_FPU_BUBBLE
BE_RSE_BUBBLE
```

Components of BE_EXE_BUBBLE

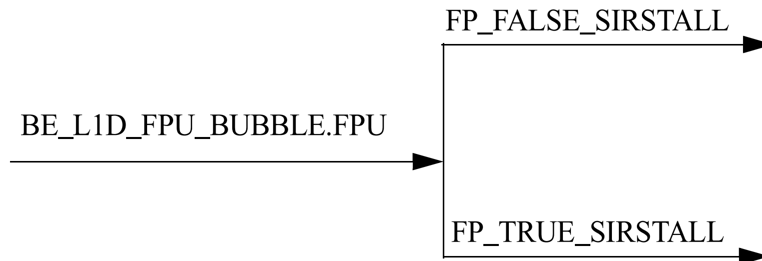


BE_EXE_BUBBLE
- BE_EXE_BUBBLE.GRALL
- BE_EXE_BUBBLE.GRGR
- BE_EXE_BUBBLE.FRALL
- L1D_READ_MISSES
- L1D_READS_SET0,1
- L2_MISSES
- L3_READS.DATA_READ.ALL
- L3_READS.DATA_READ.HIT
- L3_READS.DATA_READ.MISS

L3_MISSES
L2_REFERENCES
L3_REFERENCES
L2_OZQ_CANCELS0.ANY
L2_OZQ_CANCELS1.BANK_CONF
L2_FORCE_RECIRC.ANY
L2_FORCE_RECIRC.SAME_INDEX
L2_FORCE_RECIRC.L1W
L2_FORCE_RECIRC.OZQ_MISS
L2_FORCE_RECIRC.FILL_HIT
L2_FORCE_RECIRC.SNP_OR_L3
L2_GOT_RECIRC_OZQ_ACC
L2_ISSUED_RECIRC_OZQ_ACC
DEAR_LATENCY

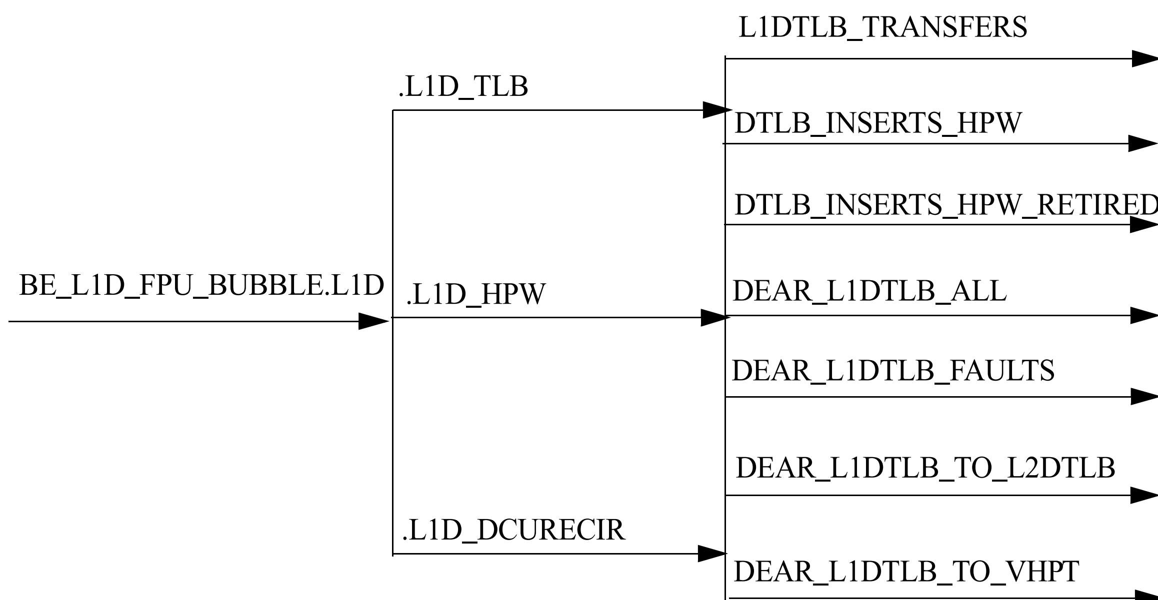
BE_L1D_FPU_BUBBLE

Contributions to BE_L1D_FPU_BUBBLE.L1D



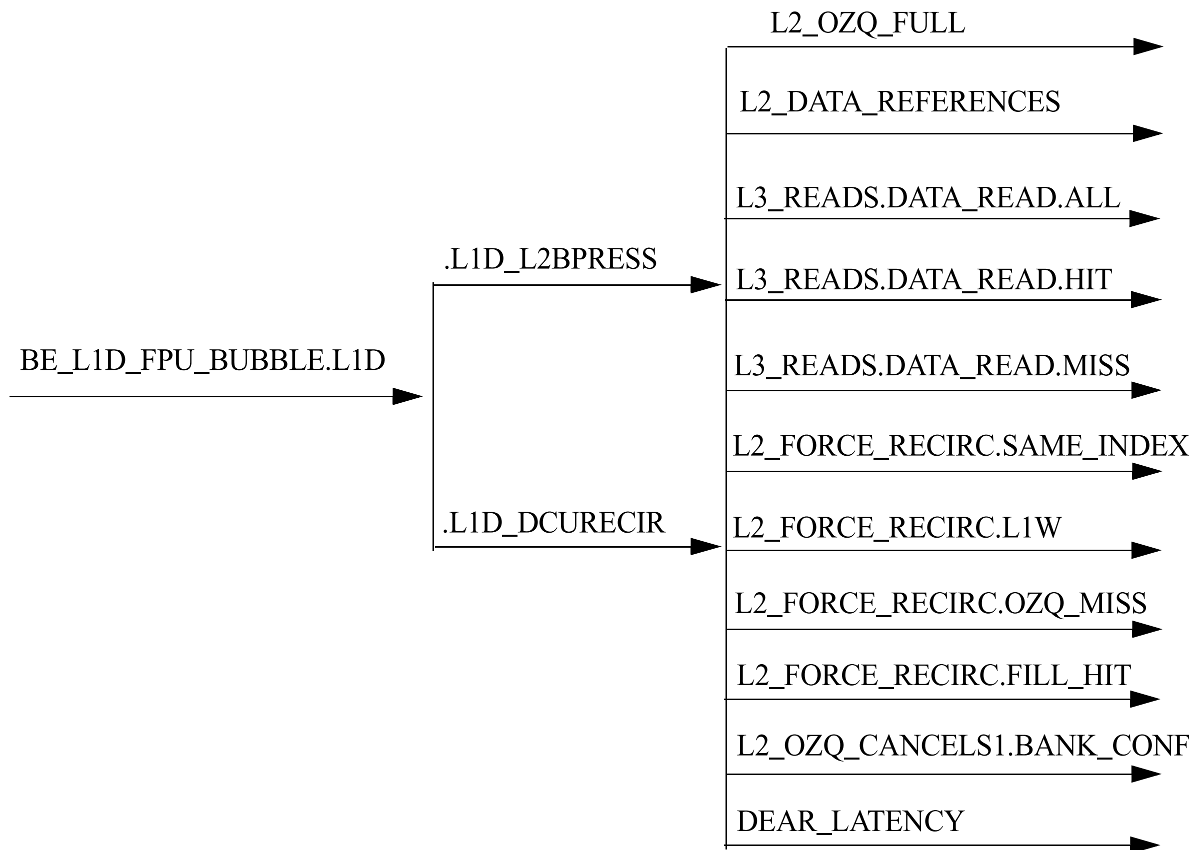
BE_L1D_FPU_BUBBLE.FPU
FP_FALSE_SIRSTALL
FP_TRUE_SIRSTALL

Contributions to BE_L1D_FPU_BUBBLLE.L1D FROM DTLB



BE_L1D_FPU_BUBBLE.L1D
BE_L1D_FPU_BUBBLE.L1D_TLB
BE_L1D_FPU_BUBBLE.L1D_HPW
BE_L1D_FPU_BUBBLE.L1D_DCURECIR
L1DTLB_TRANSFER
L2DTLB_MISSES
DTLB_INSERTS_HPW
DTLB_INSERTS_HPW_RETIRED
DEAR_L1DTLB_ALL
DEAR_L1DTLB_FAULT
DEAR_L1DTLB_TO_L2DTLB
DEAR_L1DTLB_TO_VHPT

Contributions to BE_L1D_FPU_BUBBLE.L1D_L2BPRESS




```
BE_L1D_FPU_BUBBLE.L1D_L2BPRESS
BE_L1D_FPU_BUBBLE.L1D_DCURECIR
L2_OZQ_FULL
L3_READS.DATA_READ.ALL
L3_READS.DATA_READ.HIT
L3_READS.DATA_READ.MISS
L3_MISSES
L2_REFERENCES
L3_REFERENCES
L2_OZQ_CANCELS0.ANY
L2_OZQ_CANCELS1.BANK_CONF
```

L2_FORCE_RECIRC.ANY
L2_FORCE_RECIRC.SAME_INDEX
L2_FORCE_RECIRC.L1W
L2_FORCE_RECIRC.OZQ_MISS
L2_FORCE_RECIRC.FILL_HIT
L2_FORCE_RECIRC.SNP_OR_L3
L2_GOT_RECIRC_OZQ_ACC
L2_ISSUED_RECIRC_OZQ_ACC
DEAR_LATENCY

BACK_END_BUBBLE.FE

BACK_END_BUBBLE.FE
FE_BUBBLE AND SUBEVENTS
FE_LOST_BW AND SUBEVENTS
IDEAL_BE_LOST_BW_DUE_TO_FE AND SUBEVENTS
BE_LOST_BW_DUE_TO_FE AND SUBEVENTS

BE_FLUSH_BUBBLE

ALL SUB EVENTS OF BE_FLUSH_BUBBLE
BRANCH_EVENT
BR_MISPRED_DETAIL AND ITS SUBEVENTS
BR_MISPRED_DETAIL2 AND ITS SUBEVENTS
BE_BR_MISPRED_DETAIL AND ITS SUBEVENTS

BE_RSE_BUBBLE

ALL SUB EVENTS OF BE_RSE_BUBBLE
RSE_EVENTS_RETIRED
RSE_REFERENCES_RETIRED AND ITS SUBEVENTS

Glossary

A

associativity

Number of cache lines making up an associative set

address conflicts

Conflict in addresses of data causing excess latency for access

B

bank

Memory unit, usually referring to a physical unit.

bypass

High speed data path that shortens normal delivery

C

call stack chain

Series of branches following a chain of function calls

clr hint

Clear hint..used to clear branch history table

core pipeline

Controls flow of instructions, allocates functional units as binary requires, coordinates data delivery with instruction dispatch to functional units

cycle counting

Counting CPU cycles

cycle latency

Latency measured in cycles

D

DET stage

Detection stage of core pipline. Detects exceptions and branch mispredictions and stalls in the L1D and FPU micropipelines

E

EAR events

Event Address Register Events. Intrinsicly sampling events that record instruction pointer and other aspects in hardware.

EXE stage

Core pipeline stage that dispatches instructions to functional units

EXP stage

Core pipeline stage that expands templates to select functional units.

H

Hardware Page Walker (HPW)

Part of processor architecture that looks up vertual to physical address translations in Virtual Hash Page Table

I

Instruction template

The five bits in the 128 bit instruction bundle that determine which functional units are required by each of the three 41 bit instructions. Also contains the locations of any instruction group boundaries (;; or “stop bits”) that are in that bundle.

L

leaf modules

Functions which don’t call any other functions.

M

memory subsystem

Computer subsystem that contains data. Usually consists of a multi level cache hierarchy, main memory and disk drives.

micropipelines

The Itanium® 2 processor has micropipelines to sequence the data delivery from the three caches, and sequence the long latency flaoting point and multimedia instructions.

O

OzQ

32 entry queue that controls access to L2 cache. See Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0.

P

performance monitor

Monitoring hardware inthe processor for counting the occurrences of architectural conditions. See Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0.

PMC4

See Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0.

PMD4

See Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, Revision 1.0.

Prediction history table

Table where branching history is stored to assist branch prediction.

R

REN stage

Stage of core pipeline that renames registers due to alloc instructions or register name rotation for software pipelining.

Register Stack Engine

Spills and fill general register stack to backing store as required by pipleine REN stage.

RSE Activity

Register stack engine activity.

S

sptk branch hint

Statically predicted taken branch hint.

store port

Cache access ports for storing data.

sum rule

Linear break down of a quantity.

U

umask

User mask, a bit pattern used for precise event selection

unified cache

Cache that stores both instructions and data.

V

Virtual Hash Table (VHT)

Hash encodedd table created by the operating system to allow an applications virtual addresses to be translated to physical addresses so that physical memory can be accessed through the chipset and disk system.

W

WRB stage

Pipleine write back stage. Writes results stored in registers back to data caches.

Index

A

address conflicts 31
architecture 5

C

cache
    description 9
    misses 30

core pipeline 6
counter
    Back_End_Bubble.FE 20, 71
    BE_EXE Bubble 34
    BE_EXE_BUBBLE 30
    BE_EXE_Bubble.FRALL 33
    BE_EXE_Bubble.GRALL 33
    BE_EXE_Bubble.GRGR 33
    BE_Flush_Bubble 61
    BE_Flush_Bubble.bru 61
    BE_Flush_Bubble.xpn 61
    BE_L1D_FPU_BUBBLE 30
    BE_L1D_FPU_Bubble 53
    BE_RSE_Bubble 67
    BR_MISPRED_Detail 62

cycle accounting

components 19
sum rule for Itanium 2 processors 15

D

data address conflicts 30
DET stage 7
DTLB miss 29

E

EBS 13
event
EAR 22
occurrence 21
skid 13, 22
sub 21
EXE stage 7, 29
EXP stage 6

F

FP registers 67
FPU 55
functional unit stalls 33

G

general registers 67

H

hand-coded assembler 1
Hardware Page Walker 11

I

instruction
buffer 6
streaming buffer 5

interprocedural inlining 61, 69
IPO 69

L

L1D 56
L1D_L2BPRESS 59
L1D_STBUFRECIR 60

M

memory
access stalls 29
subsystem 7

microarchitectural performance tuning 3
microbenchmarks 22

O

OzQ 10

P

physical address 11
profile guided feedback 61

Q

Qipo 69, 71
Qprof_gen 69, 71
Qprof_use 69, 71

R

references 1
REG stage 6
register stack engine 67
REN stage 6
RSE Activity 68

S

scoreboarding 55

T

translation of virtual address to physical address 11

V

virtual
    address 11
    Hash Page Table 12

VTune™ Performance Analyzer 3, 13
    limitations 14

W

WRB stage 7