

A Methodology for using the Itanium 2 Performance Counters for Bottleneck Analysis

Sverre Jarp, HP Labs
27 August 2002

Chapter 1 Background Information

1.0 Introduction

This paper is aimed at users who want to use Itanium 2 hardware performance counters when trying to identify bottlenecks in programs. The counters allow a user to characterize an application in terms of resource usage and non-usage with a high level of detail. Such characterization does not automatically warrant quick remedies for improving the performance, but it is undoubtedly a great advantage to identify the major bottlenecks, so that their removal can be initiated.

The demonstrated methodology is one of "drilling down" via more and more detailed performance counters in order to understand the performance issues in great detail. For data cache and memory stalls, the methodology identifies the portions of the memory hierarchy that contributes most significantly.

1.1 Hardware Overview

Itanium 2, just like its predecessor, comes with a complete hardware facility for performance monitoring built into the processor. The whole subsystem is described in detail in the publication "Intel Itanium 2 Processor Reference Manual for Software Development and Optimization", Intel document number 251110-001.

The subsystem is controlled by a set of registers called the Performance Monitoring Control (PMC) registers. The highlights of the performance monitoring subsystem can be understood by reviewing the capabilities of these control registers. Both Itanium and Itanium 2 have registers for three major activities:

- Counting occurrences [PMC4-PMC7]
- Establishing instruction/data event addresses [PMC10-11]
- Establishing Branch Trace Buffer [PMC12]

Other hardware registers can be used to restrict a given activity to matching op-codes or to a given set of address ranges.

In this paper we concentrate on counting occurrences, but the other two main activities will be described briefly in Appendix 1.

As far as occurrences are concerned, over 300 different counters are provided. Inside this rich set, we find counters related to:

- The CPU, such as cycle counts, instruction counts, etc
- Stalls, i.e. clock cycles where no work is being done (due to instruction latencies, cache misses, etc.)
- The Translation Look-aside Buffers (TLBs), the Advanced Load Address Table (ALAT), and the Register Save Engine (RSE)
- The cache hierarchy with its three levels (L1, L2, and L3)
- The memory subsystem, the memory bus, etc.

Since there are only four registers for counting occurrences, it is often necessary to run an executable several times in order to get a complete coverage of all the relevant performance aspects. This is another reason why the “drill down” approach, as we will encounter it in this paper, is desirable. One should always start with a set of high-level performance counters and then drill down into the areas where there is activity, but not waste time on extracting counters that reflect no significant activity.

1.2 PFMON software overview

S. Eranian (also at HP Labs) has written an easy-to-use interface to the performance monitoring hardware for Linux. There are a user command, *pfmon*, and a kernel extension, which work together to establish the performance-monitoring environment. The command accepts comprehensive user input, launches the application, and reports the results. In its simplest form an invocation may look like this:

```
pfmon -eCPU_CYCLES -us-counter-format ./hello
```

This command simply tells *pfmon* to report the total number of CPU cycles needed by the program, *hello*, and report the result in a US number format (123,456,789).

Given the richness of the performance monitoring hardware, *pfmon* has a corresponding rich set of command-line options and accepts all the counter names as defined in the Intel reference document. It should be noted that a counter described by, for instance, the name *BE_RSE_BUBBLE.all* is transformed by *pfmon* into *BE_RSE_BUBBLE.ALL*. Also note that such a name is case-insensitive, so that it can also be written as *be_rse_bubble_all*.

Furthermore, it should be mentioned that there is a good built-in help facility, *pfmon -help*, and the names of relevant counters can be found by the command: *pfmon -i substring*.

1.3 Theory and practice

To ease the understanding of the methodology of drilling down with performance counters, a program from SPECint, *crafty*, will be used as an example. It is a computer chess program. The deployed counters will therefore be the ones that demonstrate the “drill-down” methodology for this program. For the entire list of counters it is necessary to refer to the Intel document or the above-mentioned “info” feature of *pfmon*. The runs have been made on a 1 GHz HP zx2000 using the Intel ecc6 compiler under Linux (with options *-O3 -prof_gen/use*). All counts are in billions (10^9).

Chapter 2

Using the “bubble” Counters for Stall Analysis

2.1 Cycles, Instructions and CPI

The first recommendation is initially to count cycles, instructions and nops retired as well as back-end stall cycles. The corresponding invocation of *crafty* would look like this:

```
pfmon -Ecpu_cycles,ia64_inst_retired_this,nops_retired,back_end_bubble_all ./crafty < crafty.in
```

This yields the following results:

Counter name	Count (10^9)
cpu_cycles	164.4
ia64_inst_retired_this	272.9
nops_retired	52.8
back_end_bubble_all	85.2

We could have used the “raw” instruction count, *inst_dispersed*, but since it contains the count of instructions squashed because of predication or branches/interruptions, the “net” instruction count, *ia64_inst_retired_this*¹, is more useful. With *crafty*, the raw count is about 13% higher than the net count. Cycles Per Instruction, CPI, can now be computed as:

$$CPI = cpu_cycles/ia64_inst_retired_this.$$

An equally interesting ratio can be computed when “nop” instructions have been removed. In the case of *crafty*, the count of useful instructions, as we can see from the table, is 272.9 – 52.8 = 220.1.

With the information we have, we can now compute (unstalled) cycles per (useful) instruction in various flavours:

Cycles Per Useful Instruction	CPUI	164.4/220.1	0.75
Cycles Per Instruction	CPI	164.4/272.9	0.60
Unstalled Cycles Per Useful Instruction	UCPUI	79.2/220.1	0.36
Unstalled Cycles Per Instruction	UCPI	79.2/272.9	0.29

The conclusion for this analysis is that we achieve a UCPUI of 0.36 (or almost 3 useful instructions per unstalled cycle), but that stalled cycles make up more than 50% of the total. The next question is obviously: **Why?**

2.2 Stall cycles

There are five counters for the various stalls in the Itanium 2 processor back-end. The formula allowing the stalls to be subdivided, is as follows:

$$\text{Back_end_bubble_all} = \text{Be_flush_bubble_all} + \text{Be_L1d_fpu_bubble_all} + \text{Be_exe_bubble_all} + \text{Be_RSE_bubble_all} + \text{Back_end_bubble_fe}$$

The main reason why the counters are split in this way is that each one is associated with a given stage in the execution pipeline so that the corresponding counter is incremented whenever a bubble needs to be inserted into the pipeline at that particular stage. Anyway, with the four Itanium 2 counters (and remembering that we already have the total number of stall cycles), we need just one single run in order to fill the following table:

Counter name	Count (10 ⁹)	Percent
Back_end_bubble_all	85.2	100
Be_flush_bubble_all ²	11.6	13.6
Be_l1d_fpu_bubble_all	15.3	18.0
Be_exe_bubble_all	35.2	41.3
Be_rse_bubble_all	7.5	8.8
Back_end_bubble_fe	15.6	18.3

Although bubbles at the execution stage in the pipeline, *Be_exe_bubble_all*, dominate, all stall counters tend to contribute and as we will see later, they may even have to be recombined in certain ways to explain the origin of the problems that occur. For this reason, we continue by breaking up these counters into various sub-counters³.

¹ Note that this countername is an alias of IA64_TAGGED_INST_RETIREDBRPO_PMC8.

² This stall counter was computed by subtraction.

³ Note that all subcounters are computed as percentages of the TOTAL amount of stall cycles.

2.2.1 Stall counter number 1

<i>Subcounter name</i>	Count (10⁹)	Percent
Be_flush_bubble_all	11.6	13.6
Be_flush_bubble_bru	11.6	13.6
Be_flush_bubble_xpn	0.0	0.0

In most cases, this counter is practically entirely a reflection of a branch misprediction flush, which is what the suffix *bru* corresponds to. The second suffix, *xpn*, reflects much more rare occurrences of flushes caused by exceptions or interruptions. As we see in the case of *crafty* (and also frequently elsewhere), it does not contribute at all.

2.2.2 Stall counter number 2

<i>Subcounter name</i>	Count (10⁹)	Percent
Be_l1d_fpu_bubble_all	15.3	18.0
Be_l1d_fpu_bubble_l1d	15.3	18.0
Be_l1d_fpu_bubble_fpu	0.0	0.0

Also as far as the second stall counter is concerned, there is normally one key contributor (in integer programs) to the stalls, namely L1D. The reasons for L1D stalls may be further split into a rich set of finer counts in the following way.

<i>Sub-subcounter name</i>	Count (10⁹)	Percent
Be_l1d_fpu_bubble_l1d	15.30	18.0
Be_l1d_fpu_bubble_l1d_dcurecir	12.27	14.4
Be_l1d_fpu_bubble_l1d_tlb	2.64	3.1
Be_l1d_fpu_bubble_l1d_stbufrecir	0.23	0.3
Be_l1d_fpu_bubble_l1d_fullstbuf	0.07	0.1
Be_l1d_fpu_bubble_l1d_l2bpress	0.03	0.0

Without going into all the details of the memory hierarchy, let it just be said that in the case of *crafty*, the processor is having certain problems trying to load integer variables from L1D in one cycle (due to items such as recirculation of the requests and the TLB). A full understanding of the memory issues is beyond the scope of this paper, but the counters in this category are the ones that will indicate whether there are problems or not.

2.2.3 Stall counter number 3

Moving to the third global stall counter, we can obtain a detailed breakdown by using another set of sub-counters:

<i>Subcounter name</i>	Count (10⁹)	Percent
Be_exe_bubble_all	35.2	41.3
Be_exe_bubble_grall ⁴	28.2	33.1
Be_exe_bubble_frall	7.0	8.2
Be_exe_bubble_grgr	0.95	1.1
Be_exe_arcr_pr_cancel_bank	0.03	0.0

⁴ GR stands for General Register, FR stands for Floating-point Register.

This third stall counter reflects bubbles at the execution stage, mostly due to register scoreboarding. The most important counters are GRALL and FRALL, which reflect bubbles caused by inter-register dependencies (GR-GR or FR-FR) or bubbles caused by the latencies of load instructions. In the case of GR there is a second counter, which allows us to decide where the bubbles are really due to inter-register dependencies or not. This is not possible with FR. In case of doubt an inspection of the machine code in the hottest sections of the program should be performed. A ragbag of counters (related to other register, such as the Application Registers, Control Registers, Predicate Registers, etc.) can be accumulated as *Be_exe_arcr_pr_cancel_bank*⁵, but in most cases all these counts are insignificant.

In the case of *crafty*, we see that loads cause almost all GR stalls and we assume that the same is true for FR.

2.2.4 Stall counter number 4

The next counter is related to the Register Stack Engine (RSE).

Subcounter name	Count (10⁹)	Percent
Be_rse_bubble_all	7.52	8.8
Be_rse_bubble_overflow	3.56	4.2
Be_rse_bubble_underflow	3.96	4.6

This counter reflects the fact that the call stack ends up requiring more registers than the 96 integer stack registers available in the hardware. So the RSE must save (on overflow) or restore (on underflow) registers between calls. Normally the overflow is roughly equal to the underflow, but as we see from the example, they are not necessarily equal (typically because of setjump/longjump and other programming issues).

2.2.5 Stall counter number 5

The last main stall counter refers to stalls in the back-end execution engine caused by the (autonomous) front-end engine. There is no way to break down *back_end_bubble_fe* via sub-counters. What we can do, however, is to look at all the front-end stall conditions and assume that all the detrimental ones influence back-end stalls in the same proportion. There is one front-end stall counter (*fe_bubble_ibfull*), which should not be taken into account, because it indicates that the front-end stalled because the instruction buffer was full, so it simply could not do any more work. We will therefore scale the counters by dividing by *Fe_bubble_allbut_ibfull*.

Subcounter name	Count (10⁹)	Percent
Back_end_bubble_fe	15.6	18.3
Fe_bubble_all	102.2	N.A.
Fe_bubble_allbut_ibfull	50.2	N.A.

With these counters we can calculate the reduction factor as:

$$RF = (\text{Back_end_bubble_fe} / \text{Fe_bubble_allbut_ibfull}) = 0.31$$

With the scaling factor in place we measure all relevant bubble counts and scale them down in the following way:

⁵ The counter also accumulates CANCELLED loads and BANK switching.

Counter no. 5	Count (10 ⁹)	Scaled count	Percent
Back_end_bubble_fe	15.6	15.6	18.3
Fe_bubble_imiss	31.5	9.77	11.5
Fe_bubble_bubble	12.1	3.75	4.4
Fe_bubble_branch	4.74	1.47	1.7
Fe_bubble_eflush	1.93	0.60	0.7

Two other front-end bubble counters (Fe_bubble_tlbmiss and Fe_bubble_fill_recir) did not have an impact on our example.

Note that the sum of counts is slightly higher than Fe_bubble_allbut_ibfull, and this is most likely due to the fact that front-end counts seem to vary by a few percent from run to run.

2.3 Global stalls

We can now try to take a global view of the stalls by grouping relevant counts that refer to “meaningful” entities, such as branch misprediction or D-cache stalls.

High-level denominator	Formula	Count (10 ⁹)	Percent
D-cache stalls	Be_exe_bubble_grall - Be_exe_bubble_grgr + Be_L1d_fpu_bubble_L1d	42.6	50.0
Branch Misprediction	Be_flush_bubble_bru + Fe_bubble_bubble + Fe_bubble_branch	16.8	19.7
Instruction Miss stalls	Fe_bubble_imiss	9.8	11.5
RSE stalls	Be_RSE_bubble_all	7.5	8.8
FLP Units	Be_exe_bubble_frall + Be_L1d_fpu_bubble_fpu	7.0	8.2
GR scoreboarding	Be_exe_bubble_grgr	0.95	1.1
Front-end flushes	Fe_bubble_eflush	0.6	0.7
Total		85.2	100.0

So, although the counters are plentiful and complex we have succeeded in breaking the stalls into a few meaningful categories.

Most “integer” programs, from a performance point of view, behave like crafty. Typically, half of the total stall count (or more) may be attributed to the data cache or the data memory. Branch misprediction is here (and also typically) the second culprit.

In the next chapter the task will be to look at the available cache/memory counters and relate them to the D-cache stalls.

Chapter 3 Using the Memory Counters

3.1 Quick Review of the Memory Hierarchy

The Itanium 2 comes with three levels of cache, all of which are on the die itself. L2 and L3 are unified cached (i.e. shared by both instructions and data), whereas L1D is a dedicated data cache. The main characteristics are summarized in the following table:

	L1D	L2	L3
Access time	1	5+	12+
Size	16 KB	256 KB	3 MB
Line size	64	128	128
Number of lines	256	2048	24,576
Associative sets	4	8	12
Number of sets	64	256	2048
Update policy	Write-through	Write-back	Write-back
Banks	8 "groups"	16	1
Line replacement	NRU ⁶	NRU	NRU

3.2 Memory counters

The performance monitoring subsystem can report on a large number of occurrences for each cache level. In order to relate these occurrences to the stall cycles we discussed in chapter 2, we mainly focus on counting references and misses.

3.2.1 Main L2 counters

The following five counters can be used to understand the activity that takes place in L2. As usual, we give the values for *crafty*:

L2 counters	Count (10⁹)
L2_references	29.8
L2_misses	0.13
L2_data_references_L2_all	20.2
L2_inst_prefetches	5.58
L2_inst_demand_reads	5.33

It is worth noting that we cannot fit four of these counters into one *pfmon* run, because of internal hardware conflicts. Mixing in the same run, for instance, two L2 counters with two stall counters from the previous chapters, usually solves such a problem. In this way, the number of *pfmon* invocations is still kept to a minimum.

From the *crafty* results, we see that there are very few L2 misses, implying that most of the cache activity occurs in L1 and L2. By relating L2_data_references_L2_all to (L2_inst_prefetches + L2_inst_demand_reads) we see that accesses for data are about twice as frequent as accesses for instructions. We will exploit this piece of knowledge later.

Also note that the three counters just mentioned sum up to a value slightly larger than L2_references. This is probably a small inaccuracy coming from the fact that we had to obtain the results in more than one run.

3.2.2 Main L3 and TLB counters

The following four counters, which fortunately can be used together, provide relevant information about the L3 cache as well as the instruction and data Translation Look-aside Buffers (TLBs):

⁶ Not Recently Used

L3/TLB counters	Count (10⁹)
L3_references	0.177
L3_misses	- -
L2dtlb_misses	- -
l1lb_misses_fetch_all	0.009

Note that *L3_references* is higher than *L2_misses* in the previous section. This is simply due to the fact that there are asynchronous write-back operations from L2 to L3.

3.3 Formula relating memory and stall counters

In this section we provide a formula, which can help relate memory counters to stalls (bubble counters). The main purpose is to find out which part of the memory subsystem is “guilty” of producing the largest number of stall cycles. The following formula is proposed:

$$\begin{aligned}
 \text{Bubbles} = & \\
 & (\text{L2_data_references_L2_all} - \text{L2_misses_adjusted}) * N_1 \\
 & + (\text{L2_misses_adjusted} - \text{L3_missed_adjusted}) * N_2 \\
 & + \text{L3_misses_adjusted} * N_3 \\
 & + \text{L2dtlb_misses} * N_4
 \end{aligned}$$

The formula simply stipulates that each cache/TLB level contributes a number of bubbles which correspond to the number of data references satisfied at that level in the hierarchy times an “average” number of cycles. The reason we need to adjust *L2_misses* and *L3_misses* comes from the fact that they also include activity caused by instruction fetching or prefetching. Consequently, we use the ratio we calculated for the L2 accesses in section 3.2.1 (which was 2/3 for *crafty*) to adjust these misses.

The cycle counts are estimated as follow:

Component	L2	L3	Memory	D-TLB
Access time	5+	12+	110 - 190	32
Proposed factor	$N_1 = 2$	$N_2 = 10$	$N_3 = 150$	$N_4 = 30$

For L2 and L3 we propose a cycle count, which is smaller than the access time, because the compiler tends to hide some of the latency when performing the instruction scheduling. For the memory latency, which can be somewhat unpredictable according to the access pattern and access frequency, we simply propose to use the average. For the L2DTLB misses, we propose to use the (rounded) access time.

3.4 Applying the formula to *crafty*

For *crafty* the formula degenerates to a very simple calculation because of the high ratio *L2_references/L2_misses* (~230). Practically all the data needed by *crafty* fits in L2 cache, and the resulting computation becomes:

$$\begin{aligned}
 \text{Bubbles} = & \\
 & (\text{L2_data_references_L2_all}) * N_1 = 20.2 * 2.0 = \\
 & 40.4
 \end{aligned}$$

The “correct” number is 42.6 so we see that we are rather close.

Since the example turned out to be rather trivial, we may want to test the formula on a more complex example from the SPECint suite.

3.5 Applying the formula to *mcf*

This second SPECint program, *mcf*, is an application written at the Konrad-Zuse-Zentrum Berlin and used for single-depot vehicle scheduling in public mass transportation. In the data layout we find large link-lists of structures, which stress all levels of the memory hierarchy. First of all, we read out the basic performance counters, as we did for *crafty* in chapter 2.

Counter name	Count (10⁹)
CPU_cycles	37.0
Back_end_bubble_all	30.0
Be_exe_bubble_grall	23.6
Be_exe_bubble_grgr	- -
Be_L1d_fpu_bubble_all	5.9

The total number of D-cache stalls is computed via the formula:

$$\begin{aligned} \text{D-stalls} = \\ (\text{Be_exe_bubble_grall} - \text{Be_exe_bubble_grgr} + \text{Be_L1d_fpu_bubble_L1d}) = \\ 29.5 \end{aligned}$$

We can immediately draw a couple of conclusions. Stall cycles make up 81% of the total CPU cycles and D-cache stalls account for almost 99% of the stalls.

A new set of measurements gives us the relevant cache counters;

Cache counters	Count (10⁹)
L2_references	1.79
L2_data_references_L2_all	1.78
L2_misses	0.62
L3_misses	0.094
L2dtlb_misses	0.15

First of all, it is important to note that there is no need to adjust *L2_misses* and *L3_misses*, since *L2_references* is entirely dominated by data references. The total formula becomes:

$$\begin{aligned} \text{Bubbles} = \\ (\text{L2_data_references_L2_all} - \text{L2_misses}) * 2 \\ + (\text{L2_misses} - \text{L3_misses}) * 10 + \text{L3_misses} * 150 + \text{L2dtlb_misses} * 30 = \\ (1.78 - 0.62) * 2 + (0.62 - 0.094) * 10 + 0.094 * 150 + 0.15 * 30 = \\ 2.2 + 5.3 + 14.1 + 4.5 = 25.9 \end{aligned}$$

The computed number, 25.9, is 12% lower than the number we expected, 29.5. One reason may be that the memory accesses, which are relatively unstructured in *mcf*, due to the pointer chasing through the link-lists, take more than 150 cycles. In any case, we are able to draw a couple of significant conclusions:

- Memory access delays account for approximately 50-55% of the total stalls
- DTLB stalls may account for about 15-17% of the stalls
- L2 and L3 hits account for the remaining portion of bubbles (~30%)

Chapter 4

Conclusions

Performance counters are extremely useful when trying to understand detailed behavior of a computer program by allowing users to drill down into problem areas. In this paper we have combined three categories of counters in order to develop a methodology for drilling down to allow a comprehensive performance-related characterization of an application. These three categories of counters are.

1. CPU cycles and instruction counts, which enable us to compute the various “cycles per instruction” measures to get an overall picture of how well the program is running
2. Stall counters, which enable us to understand the nature of the stalls suffered by the application. Stalls can be regrouped as a set of easily understandable entities: D-cache stalls, Branch Misprediction stalls, I-cache stalls, RSE stalls, FLP stalls, GR scoreboarding stalls, and front-end flush stalls. The first two are generally the dominant ones.
3. Memory counters, which can be successfully used to understand the various contributions to the total D-cache stalls experienced by an application, even if the formula used is only approximately correct.

The fact that there are only four counters available in the hardware (and the fact that some counters cannot be combined with each other in the same run) can make the data collection somewhat tedious for long-running programs. On the other hand, the counters display great resilience so that multiple runs do not seem to cause accuracy problems between runs.

Appendix 1

Certain Advanced *pfmon* Options

A1 Housekeeping Options

A1.1 Choosing kernel-/user-level data collection

--kernel-level [-k]
--user-level [-u]

These options allow data collection to be done at kernel-level or user-level. Note the shorter abbreviations, -k/-u, which may be used.

A1.2 Requesting system-wide data collection

--system-wide
--aggregate-results

The first option is useful when you want to collect *pfmon* statistics across a multi-processor. The second option instructs *pfmon* to return one set of aggregated counters (rather than sets of counters per processor)

A1.3 Establishing address limitations

--drange=*start,end*
--irange=*start,end* [--irange=*myroutine*]
--inverse-irange
--trigger-address=*addr* [--trigger-address=*myroutine*]

The first two options are useful to constrain *pfmon* inside an address range, either for data or for instructions. When a routine name is used for *irange*, *pfmon* will use the symbol table to figure out the start and the ending address of the routine.

The third option inverts the instruction-range restriction, i.e. requests *pfmon* to operate on the addresses outside of the restricted range.

The fourth option delays monitoring until the specified address has been reached

A1.4 Setting the duration of a session

--session-timeout=*secs* [-t *secs*]

This option allows the user to limit the duration of the data-collecting session.

A1.5 Writing results to a file

--outfile=*xxxxxxxxxx*
--append

This option tells *pfmon* to write the results to a file. The second option can be added if *pfmon* should append to the output file.

A2 Advanced Functionality

A2.1 Matching op.codes

```
--opc-match8=val  
--opc-match9=val
```

With these options the user can limit the results to certain op-codes. This can be very useful when it is necessary, for instance, to know the number of instructions executed of a certain type, such as an *xma* instruction.

The register value needed as input is a combination of 28-match bits (from the instructions) and a 28-bit mask (plus some other control bits). Please refer to chapter 10.3.4 in the Optimization Guide for more information.

Note, that certain instruction categories can be counted directly:

- LOADS_RETIRE
- STORES_RETIRE
- NOPS_RETIRE
- BR_MISPRED_DETAIL_ALL_ALL_PRED (All branches)

A2.2 Using Event Address Registers

The Performance Monitoring Unit allows five kinds of events to be sampled. These events are:

- Instruction cache misses
- Instruction TLB misses
- Data cache load misses
- Data TLB misses
- ALAT misses

Once again, it is recommended to study the corresponding details of correct usage in the Optimization Guide. We will just show an example of how *pfmon* can be instructed to record cache misses, for instance:

```
-eDATA_EAR_CACHE_LAT4 --long-smpl-period=nnnn --smpl-outfile=xxxxx --no-smpl-header
```

With these options, *pfmon* will sample data cache misses, which have a latency of 4 (which implies any latency). The sampling period would be *nnnn* and the results would be written to a special output file *xxxxx* without a header. It would then be suitable for post-processing by an analysis program.

A2.3 Using Branch Trace Buffer

The Performance Monitoring Unit also allows branches to be captured according to certain criteria, such as correctly/incorrectly predicted branch path or branch target. Here is a simple example:

```
-eBRANCH_TARGET --btb-ppm-incorrect --long-smpl-period=nnnn
```

With these options, *pfmon* will capture branches for which the path is predicted incorrectly. Other BTB options can be found in the *pfmon* help output.

Appendix 2 Work Sheet

Instructions

	RUN 1	RUN 2	RUN 3
IA64_INST_RETIRED_THIS			
NOPS_RETIRED			

Cycles

CPU_CYCLES			
BACK_END_BUBBLE_ALL			
UNSTALLED (Take the difference)			

Possible back-end stalls (1 – 5)

BE_FLUSH_BUBBLE_ALL			
BE_FLUSH_BUBBLE_BRU			

BE_L1D_FPU_BUBBLE_ALL			
BE_L1D_FPU_BUBBLE_L1D			
BE_L1D_FPU_BUBBLE_L1D_DCURECIR			
BE_L1D_FPU_BUBBLE_L1D_TLB			
BE_L1D_FPU_BUBBLE_L1D_STBUFRECIR			
BE_L1D_FPU_BUBBLE_L1D_FULLSTBUF			
BE_L1D_FPU_BUBBLE_L1D_L2BPRESS			

BE_EXE_BUBBLE_ALL			
BE_EXE_BUBBLE_GRALL			
BE_EXE_BUBBLE_GRGR			
BE_EXE_BUBBLE_FRALL			

BE_RSE_BUBBLE_ALL			
BE_RSE_BUBBLE_OVERFLOW			
BE_RSE_BUBBLE_UNDERFLOW			

BACK_END_BUBBLE_FE			
------------------------------------	--	--	--

6 possible front-end stalls (with negative impact)

FE_BUBBLE_ALLBUT_IBFULL			
FE_BUBBLE_FEFLUSH			
FE_BUBBLE_TLBMIS			
FE_BUBBLE_IMISS			
FE_BUBBLE_BRANCH			
FE_BUBBLE_FILL_RECIR			
FE_BUBBLE_BUBBLE			

Memory

L2_REFERENCES			
L2_DATA_REFERENCES_L2_ALL			
L2_INST_PREFETCHES			
L2_INST_DEMAND_READS			

Further memory

L2_MISSES			
L3_REFERENCES			
L3_MISSES			

TLB

L2DTLB_MISSES			
ITLB_MISSES_FETCH			