Tuning Dynamic Branch Prediction on ADSP-BF70x Blackfin+™ Processors

Contributed by Manali Vispute

Rev 1 – July 8, 2015

Introduction

The fast execution rate of ADSP-BF70x Blackfin+™ processors is in a large part due to its ten-stage instruction pipeline. The flipside of having such a deep pipeline is that branch instructions require that the pipeline be flushed, and the new instruction fetched from the branch target address must traverse the entire pipeline before it can be executed. This results in undesired latency in the execution of such branch instructions.

Branches can be accelerated if the processor correctly predicts the target of an upcoming branch instruction and automatically begins fetching instructions from that branch target's address. This is referred to as branch prediction, which can be static or dynamic in nature. Static branch prediction depends on information gathered before the execution of programs, while dynamic branch prediction depends on such information gathered during run-time.

Branch Predictor Overview

The ADSP-BF70x processors have a dynamic branch predictor (BP) unit. Once enabled, it can significantly reduce the latency associated with branches in a program. When the BP is disabled, the ADSP-BF70x processors rely only on static prediction to accelerate the branches and keep the instruction pipeline's utilization high.

The ADSP-BF70x processors also feature a Performance Monitor Unit (PMU) that monitors the processor's internal resources. It can count a set of processor events during program execution. In this EE-note, the PMU is used to monitor the performance of the BP. For a detailed description of the BP and PMU, please refer to the *ADSP-BF70x Blackfin+ Processor Programming Reference*[1].

The BP contains a 16-kbit RAM, where learned predictions are stored. This RAM can only be accessed once per cycle, so heuristics control whether the RAM is accessed in order to learn new information about a branch or to predict the target of a branch. These heuristics can be influenced by settings in the `BP_CFG` register. Different programs have different characteristics, and the best heuristic for your program may not be the default setting; therefore, it may be possible to improve performance by exploring alternate settings.

In ADSP-BF70x processors, the sequencer controls the program flow by providing the address of the next instruction to be executed. The BP runs ahead of the sequencer and predicts the target address of branch instructions in a program. This prediction is dynamic, based on the location and direction of branches previously executed by the processor. In this manner, the BP predicts the flow of control in a program and keeps the instruction pipeline's utilization high.

During each instruction fetch cycle, the fetch unit fetches one line (64 bits) of instruction data, and the BP is capable of predicting two branches within each line. The BP table in the 16-kbit RAM can be viewed as two-way set associative, where

Set0 is associated with the first branch learned in a line, and Set1 is associated with the next branch learned in the same line. Each set stores the type, source address, and target address of the static branches learned by the BP. For conditional branches, an additional four-value prediction code is also stored. The prediction code indicates the direction the branch is likely to take based on execution history. For the new branches learned, the predictor overwrites one of the two sets using a Least Recently Used (LRU) policy. In this policy, the LRU bit is used to point to the oldest branch table entry that was accessed for each line. This entry will be overwritten when a new branch is to be learned.

The BP performs two kinds of accesses to the BP table:

- fetch accesses - reads table entries for determining possible branch hits
- management accesses – adds/updates table entries

While a fetch access is performed every time an instruction is fetched from memory, a management access is performed only in those cycles where instruction fetches are not made. Thus, the BP gives precedence to branch prediction over adding or updating entries in the BP table.

The sequencer issues four types of requests/accesses to the BP table, as shown in Table 1:

Access Name	**Access Description**
Learn	Creates new entry in the BP Table
Update	Updates the prediction value in a table entry
Instruction Mispredict	Occurs when the type of a prediction does not match the type expected by the sequencer. Prevents further predictions from that entry.
Address Mispredict	Occurs when the target address of a prediction does not match the address expected by the sequencer. Updates the target address of the table entry.

Table 1. BP Table Accesses

The sequencer provides the data for these requests at two different points in the instruction pipeline; thus, the BP needs to buffer the data coming from the sequencer so that it can be written to the BP table in a single access. Also, the strategy of performing management accesses in between instruction fetches calls for the buffering of data before it can be entered into the BP table. This is done using two data store buffers, Store Buffer 0 and Store Buffer 1, which are used to store requests from the sequencer before they are written to the BP table.

The store buffers can be in any of three states:

- idle
- waiting for additional sequencer data
- full

When a store buffer gets all the data from the sequencer, it goes from the waiting state to the full state. Once full, the store buffer moves its contents to the table control buffer of the BP table and goes back to the idle state. The table control buffer then waits for a non-fetch cycle to write its contents to the BP table. In order to ensure that this wait is not indefinite, the number of sequential instruction fetch cycles are counted, and if they exceed a certain threshold, the sequencer is requested to hold off an instruction fetch for one cycle. The number of sequential fetches counted are stored in the STMOUTCNTR field of the BP_STAT register, while the threshold can be set using the STMOUTVAL field of the BP_CFG register.

The store buffers interact in two ways. The first is the order in which the store buffer requests are sent to the table control buffer for execution, and the second is the order in which the sequencer requests are loaded to the buffers. The order in which requests are sent to the table control buffer is straightforward, as the most recently loaded or newest store buffer request is sent to the table control buffer first. If a request has already been asserted (but not accepted) by the table control buffer, the request will be de-asserted and the newer request will be asserted. Once the new request is asserted by the table control buffer, the old request may be reasserted. This policy ensures that branches close to the current PC are loaded into the table as quickly as possible, increasing the probability of branches being predicted in tight loops.

On the other hand, the loading of store buffers with the sequencer requests is controlled by several policies. The first policy is that Store Buffer 0 will always be loaded first after reset. The second is that when a store buffer is in the full state, the other buffer is chosen by the sequencer to write to. In the case where none of the requests have been executed to the table before a new request is received, the contents of the oldest store buffer will be overwritten. The third policy is that store buffers which have their data accepted by the table control buffer will be loaded next.

Default BP Configuration

Dynamic branch prediction in ADSP-BF70x processors is enabled by default in the SYSCFG register. Once enabled, the BP starts executing immediately. However, it requires 132 core clock initialization cycles before it can start improving the latency of branches. During this initialization period, all the table entries are written with 0s, one row at a time. All other types of accesses to the table are blocked, so no predictions, learning, or updates will be performed by the BP during this time.

The predictor can be configured using the BP_CFG register, which contains the enable bits for dynamic prediction of each type of branch supported by the BP. When enabled, the BP will learn new branches of this type and add them to the table. The STMOUTVAL field of the BP_CFG register can be used to set the threshold on the number of sequential instruction fetches.

The reset value of the BP_CFG register enables prediction for all types of branches except for the JUMP Condition Code (JUMPCC) branch, and the STMOUTVAL field is set to 22. However, the C run-time code featured in the CrossCore® Embedded Studio 2.0.0 development tools automatically reconfigures the BP_CFG register to enable all branch prediction and the Skip Update LRU Mode. Additionally, the buffer timeout is reduced to 0, and the BP table is flushed.

While this configuration yields good performance for most applications, it may not be optimal in all cases. See the *Code Example* section for assistance determining the best configuration for your application.

Guidelines for BP Tuning

If cycle count reduction in a program is desired, then tuning the BP may help. The following guidelines are recommended.

First, the store timeout can be set to a value lower than the reset default to avoid buffer overwrites before the sequencer request can be executed to the BP table. Experiment with different values in the STMOUTVAL field of the BP_CFG register to determine the best configuration for the program.

Second, either the Skip Update or Skip Update LRU Mode can be enabled to reduce the number of stalls incurred due to table updates. These modes can be enabled using the SKUPD and SKUPDLRU bits of the BP_CFG register, respectively. The Skip Update Mode causes updates to be skipped when the prediction code for a branch is either strongly taken or strongly not taken. The Skip Update LRU Mode also causes updates to be skipped if the prediction code for a branch is either strongly taken or strongly not taken and the predicted branch is not the oldest in the table for a specific instruction line. This keeps the recently accessed branches in the table for a longer period of time, thus increasing the frequency of predictions near the program counter (PC).

Finally, it is good programming practice to reset the BP if self-modifying code is used. This will clear invalid branch entries associated with previous execution from the affected memory space from the BP table and result in correct operation. For a detailed description of the procedure for clearing the BP table, see the *ADSP-BF70x Blackfin+ Processor Programming Reference*.

Code Example

The *associated ZIP file*[3] contains example code providing insight into how the BP settings affect the performance of the program and how it can be tuned to improve the performance. The example was implemented using the CrossCore® Embedded Studio 2.0.0 (CCES 2.0.0) development tools and the ADSP-BF707 EZ-Board™ evaluation platform[2].

In this example, the BP performance is evaluated by counting the number of processor cycles required to execute a loop. The BP events that occur while the loop is executing are monitored using the PMU. These events are learn requests from the sequencer and branches learned by the BP table.

The code calls the BP_Test() function four times. Each call to this function sets the BP_CFG register to some value and flushes the BP table. Once the BP table is cleared, all branches have to be learned anew. During each run of the BP_Test() function, the stringlength() function is called twice.

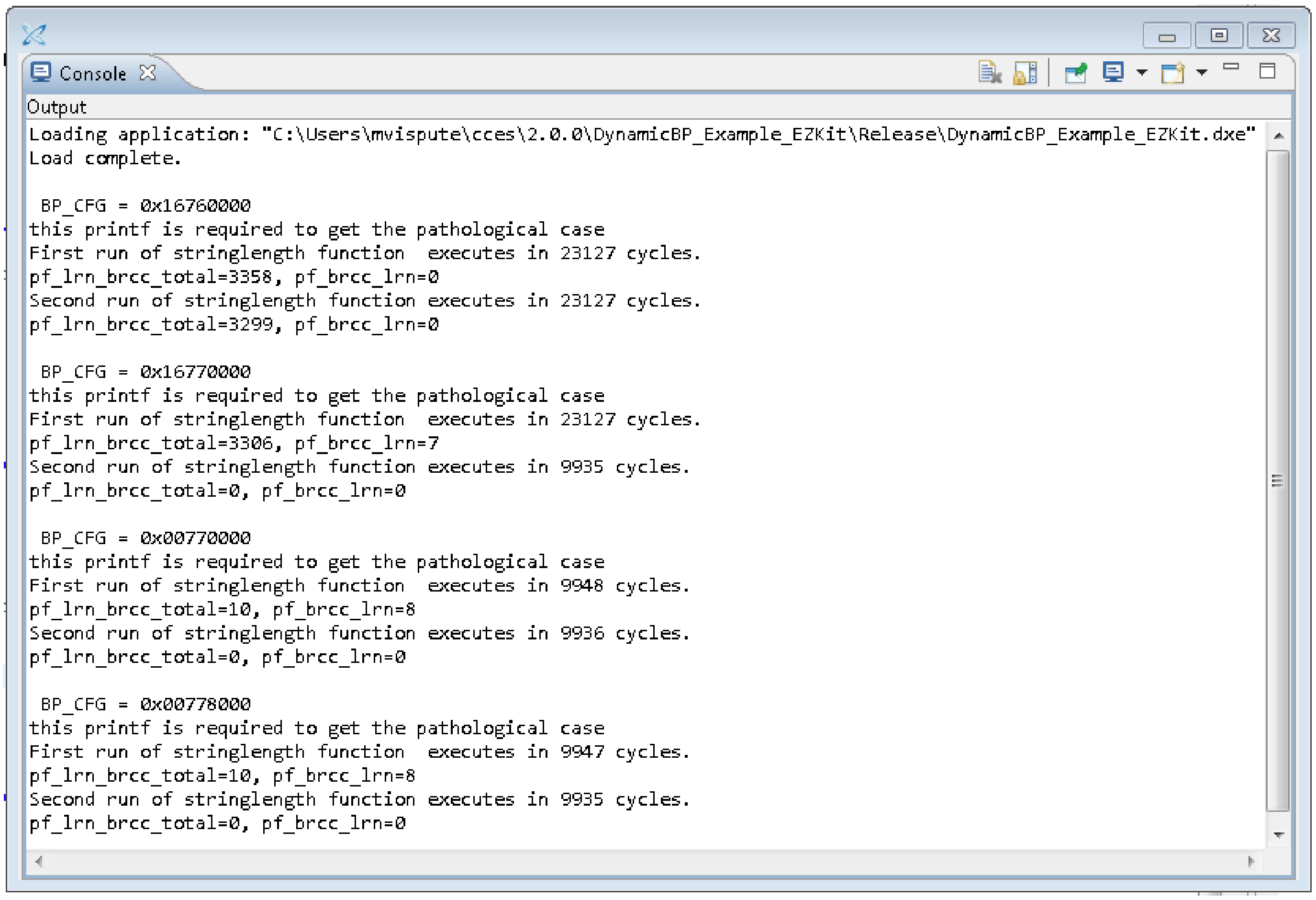
As the name suggests, the stringlength() function determines the length of a string passed to it. In this case, the string is 3299 characters long. The function consists of a tight three-instruction loop, as follows:

```
    _looptop:
11a00ad8:  R0=B[P0++](Z);
11a00ada:  CC=R0==0x0;
11a00adc:  IF!CC JUMP -0x4(BP);
```

The loop has a conditional jump branch, where the BP modifier is used to initialize the dynamic branch predictor.

The clock() function in the example code is used to count the number of processor cycles required to execute the loop. In order to ensure that the loop does not execute during the BP table initialization period, the csync() function is called before calling the stringlength() function. The csync() function ensures resolution of all pending core operations and flushing of the core store buffer before proceeding to the next instruction.

Since the string passed to the stringlength loop is 3299 characters long (including the terminating 0), the loop should execute in slightly more than 9894 cycles, provided that the conditional jump at the end of the loop is correctly predicted. However, it is observed that the stringlength loop executes in anywhere from 9935 to 23,127 cycles, depending on the BP_CFG register settings, as depicted in Figure 1.



```
Loading application: "C:\Users\mvispute\cces\2.0.0\DynamicBP_Example_EZKit\Release\DynamicBP_Example_EZKit.dxe"
Load complete.

 BP_CFG = 0x16760000
this printf is required to get the pathological case
First run of stringlength function  executes in 23127 cycles.
pf_lrn_brcc_total=3358, pf_brcc_lrn=0
Second run of stringlength function executes in 23127 cycles.
pf_lrn_brcc_total=3299, pf_brcc_lrn=0

 BP_CFG = 0x16770000
this printf is required to get the pathological case
First run of stringlength function  executes in 23127 cycles.
pf_lrn_brcc_total=3306, pf_brcc_lrn=7
Second run of stringlength function executes in 9935 cycles.
pf_lrn_brcc_total=0, pf_brcc_lrn=0

 BP_CFG = 0x00770000
this printf is required to get the pathological case
First run of stringlength function  executes in 9948 cycles.
pf_lrn_brcc_total=10, pf_brcc_lrn=8
Second run of stringlength function executes in 9936 cycles.
pf_lrn_brcc_total=0, pf_brcc_lrn=0

 BP_CFG = 0x00778000
this printf is required to get the pathological case
First run of stringlength function  executes in 9947 cycles.
pf_lrn_brcc_total=10, pf_brcc_lrn=8
Second run of stringlength function executes in 9935 cycles.
pf_lrn_brcc_total=0, pf_brcc_lrn=0
```

Figure 1. Console Output from Example Program

In this example, the PMU is used to monitor two BP events - branch learn requests from the sequencer (pf_lrn_brcc_total) and branch learn requests written to the BP table (pf_brcc_lrn). For a detailed description of the PMU events, please refer to the *ADSP-BF70x Blackfin+ Processor Programming Reference*.

At first, the BP_CFG register is the reset value 0x16760000. The STMOUTVAL is set to 22, and prediction for all branches except for conditional jump is enabled. Both runs of the stringlength() function take 23,127 cycles. The only differences are the number of learn requests from the sequencer and the number of branches learned by the BP. During the first run of the stringlength() function, the PMU counts the number of conditional branch learn requests from both the stringlength() and printf() functions. There are 3358 requests from the sequencer to learn a conditional branch, but none are actually learned by the BP due to the fact that prediction for conditional jumps is disabled. During the second call to the stringlength() function, the PMU only counts the conditional branch learn requests from the stringlength() function; hence, the number of learn requests is 3299 and, again, no conditional branches are learned by the BP. Since the number of cycles taken to execute the loop is much more than expected, the BP must be tuned.

Because there are conditional jumps in the example code, the first step in tuning the BP is to enable the prediction of the conditional jump branches. As such, the BP_CFG register is set to 0x16770000. The first call to the stringlength() function again takes 23,127 cycles, even when the prediction of conditional jump branches is enabled. It is observed that there are 3306 learn requests from the sequencer, but the branches are not learned until very late in the execution. However, during the second call to the `stringlength()` function, the number of cycles goes down to 9935, and the number of learn requests and branches learned is 0. This indicates that branches are already learned and present in the BP table.

The inability of the BP to learn branches is explained by the fact that the `stringlength()` loop iterates (loops back) every seven cycles. The ADSP-BF707 fetch unit issues a series of 10 consecutive fetches, which is restarted by a new loopback on every 8th cycle. These continuous fetches starve the predictor of the cycles required to create or update a branch entry. As a result, the table control buffer gets overwritten before it can write its contents to the BP table, and the predictor fails to learn the branch despite executing the loop several times.

To avoid buffer overwrites, the store timeout value must be reduced. In the second step, the `STMOUTVAL` is reduced to 0, which reduces the number of processor cycles to 9948 and 9936, respectively. Programmers should experiment with different values of `STMOUTVAL` to arrive at the best configuration.

The last step in tuning the BP is to enable the Skip Update LRU mode by setting the `SKUPDLRU` bit of the `BP_CFG` register. This mode reduces the number of stalls due to table update. The number of processor cycles required is 9947 and 9935, respectively. Likewise, the Skip Update Mode can also be enabled to further reduce the number of stalls, but the Skip Update LRU Mode is preferred because it yields better LRU handling, which increases the frequency of predictions near the program counter (PC).

As a result of the BP optimization steps taken (summarized in Table 2), the best setting for the `BP_CFG` register in the example code is 0x00778000. In CCES 2.0.0, the start-up code sets the default configuration of BP to 0x00778001. It sets the store timeout to 0, enables the Skip Update LRU Mode, and flushes the BP table.

BP Optimization Steps	Cycles		Branch Learn Requests		Branch Learns	
	1st Run	2nd Run	1st Run	2nd Run	1st Run	2nd Run
1. Enable Conditional Jump Branch Prediction	23127	9935	3306	0	7	0
2. Reduce Store Timeout to 0	9948	9936	10	0	8	0
3. Enable Skip Update LRU Mode	9947	9935	10	0	8	0

Table 2. Performance Measurements during BP Tuning

Conclusion

This EE-Note discussed how to tune the ADSP-BF70x Blackfin+ processor's dynamic branch predictor and provides example code for programmers to gain insight into the effects of BP settings on the application's performance.

While the default BP configuration established by the CCES 2.0.0 run-time code results in good performance for most applications, there is no universal best setting. In many cases, better performance can be achieved by manipulating the BP configuration settings through empirical testing of the embedded application code.

References

[1] *ADSP-BF70x Blackfin+ Processor Programming Reference.* Rev 0.2, May, 2014. Analog Devices, Inc.

[2] *ADSP-BF707 EZ-KIT Lite Evaluation System Manual*. Rev 1.0, May, 2014. Analog Devices, Inc.

[3] *Associated ZIP File (EE373v01.zip) for Tuning Dynamic Branch Prediction on ADSP-BF70x Blackfin+ Processors (EE-373).* July 2015. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 1 – July 8, 2015 *by Manali Vispute*	Initial Release