Dynamic Branch Decoupled Architecture

Akhilesh Tyagi, Hon-Chi Ng*
Dept. of Electrical & Computer Engg.
Iowa State University
Ames, IA 50011
tyagi, hcng@iastate.edu

Prasant Mohapatra
Dept. of Computer Science & Engg.
Michigan State University
E. Lansing, MI 28824
prasant@cse.msu.edu

Abstract

We propose an alternative approach to branch resolution based on the earlier work on decoupled memory architectures. Branch decoupling is a technique to decouple a single instruction stream program into two streams. One stream is solely dedicated to resolving branches as early as possible (both the branch condition and the branch target). The resolved branch targets are consumed by the other computing stream through a queue. We have proposed a compiler based, static branch decoupling methodology earlier. In this paper, we propose a dynamic branch decoupled (DBD) architecture. Simulations show a speedup of 25.6% *for SPEC95 integer benchmarks and* 6.1% *for SPEC95 FP benchmarks over a 2-level adaptive branch predictor. The average number of branch penalty cycles per instruction for DBD reduces to* $.0475$ *compared to* $.0835$ *for the 2-level branch predictor.*

1 Introduction

The instruction-level-parallel processors rely upon dynamic branch prediction techniques [11] to improve performance. Several attempts have been made in the recent years to improve the accuracy of branch prediction [3, 7, 12]. These predictors try to adapt to the dynamic program behavior in order to improve their performance. With typical branch frequency of about 20%, the impact of branches on pipeline performance is quite significant.

In this paper, we propose the dynamic branch decoupled (DBD) architecture built upon the static, branch decoupled architectures [10]. The primary technique is to dynamically decouple the incoming, single instruction stream into two streams. One instruction stream is dedicated to resolving the branch instructions which are executed on a branch processor (BP). The branch outcomes are conveyed through a queue to the other instruction stream executing on program processor (PP). The instruction fetch engine of the program processor uses the addresses in this queue for the control flow. Note that our approach is not to improve the prediction accuracy, but to eliminate the need for predictions by resolving the branches ahead of time, whenever possible.

The proposed DBD architecture is modeled using the SimpleScalar toolkit [2]. We have used the SPEC95 benchmark suite to evaluate the performance of the DBD architecture. Since branch prediction, specifically 2-level adaptive prediction, is widely deployed to minimize impact of control hazard, it is used as the base comparison point for the DBD architecture. The performance of branch prediction, as measured in prediction accuracy, depends on how closely related the branch outcomes are, whereas the DBD architecture depends on how loosely coupled the basic blocks are, whose performance is measured in branch-outcome availability. We simulated three of the SPEC95 integer benchmarks and three of the SPEC95 FP benchmarks on a DBD simulator. These benchmarks were also simulated with the out-of-order SimpleScalar simulator using a 2-level adaptive branch predictor with single 8-bit history shift register at 1st level and 1k 2-bit counter entries at 2nd level along with a 8K BTB of 512 sets with 4-way set associativity. The DBD simulations were performed along a wide range of the architecture queue parameters. The DBD speedup over branch prediction was as high as 41% and as low as 1.3%. The average speedup was 26.6% for the integer benchmarks and 5.24% for the FP benchmarks. The average speedup over all the six benchmarks (`go, li, m88ksim, swim, applu, fpppp`) was 15.97%. The branch penalty cycles also declined significantly. The average number of branch penalty cycles per instruction for DBD reduces to $.0475$ compared to $.0835$ for the 2-level branch predictor.

The rest of the paper is organized as follows. In the next section (Section 2, we describe some of the previous work related to the DBD architectures. Section 3 describes the static branch decoupled architecture. The proposed dy-

*This work was partially supported by NSF Grant #MIP9703702.

namic adaptation of this architecture, DBD architecture, is described in Section 4. The experimental setup and results are given in Section 5. We conclude in Section 6.

2 Related Work

Decoupled architectures have been proposed earlier in the context of decoupling access and execution [9, 8, 6]. In addition to pipelining, the concurrent processing of execution and memory access can result in significant performance improvement. These techniques form the basis of the decoupled access/execute architectures. These architectures use two processors, one to perform address calculations and load and store operations, and the other to operate on the data and produce results. The two processors are usually termed as access processor and execute processor, respectively. FIFO buffers or queues are provided between the two processors to maximize the overlap and independence of the two processors. This implementation allows the access processor to *slip* ahead of the execute processor and fetch data before the execute processor needs it. Goodman *et al.* [5] describes a pipelined implementation of a memory decoupled architecture. This study explores the potential speedup achievable from a decoupled architecture. The average speedup over 12 Lawrence Livermore loops was 1.58 with several loops achieving a speedup of close to 2. An innovative integration of decoupled architectures and loop pipelining is reported in Bird [1]. The memory access processor is further decoupled into a control processor and an address processor. The control processor is assigned the task of recognizing basic blocks for the loop pipelining accounting for flushing of a loop pipeline. A study of the memory latency effects in decoupled architectures has been reported in [6]. A static branch decoupled architecture was proposed in [10], where instructions are decoupled into two separate streams, namely program stream and branch stream. A compiler performs the dependence analysis and splits the instructions into P-stream and B-stream. The static decoupling has binary compatibility problem and cannot handle branch dependencies that are not visible at compile time.

3 Branch Decoupled Architecture

The branch decoupling works under the premise that the dependences for a branch within a basic block are shallower than the dependences for the rest of the basic block computations. The primary approach is to statically decouple a program into two instruction streams, a branch stream (B-stream) – whose sole responsibility is to resolve branches, and a program stream (P-stream) – consisting of all the leftover computations. The P-stream does not contain any branch instructions in a fully decoupled program. The control flow is left entirely in the B-stream. The architecture consists of two logical processors, branch processor (BP) and program processor (PP) (Figure 1 (a)). BP has a traditional program counter, and can even undertake branch prediction. However, the program processor's program counter (PPC) is a queue (Figure 1 (b)) of address offsets and some additional information provided by the branch processor. A branch instruction in the B-stream has sufficient information about the basic blocks controlled by it both in the B and P streams. For instance, `BEQZ R1, B-Offset, P-Offset, taken-count, not-taken-count` is a possible branch instruction in the B-stream. It evaluates the branch condition based on a register in its own register file, `R1`. If the branch is taken, the B-stream PC (BPC) gets the BTA given by adding the current BPC value to the PC-relative offset `B-Offset`. At this point the information about the P-stream basic block controlled by this branch is also queued into the PPC queue, the PC-relative offset for the PPC `P-Offset` and the instruction count of the taken P-stream basic block `taken-count`. If the branch is not taken, the BPC continues sequentially. However, the PPCQ gets an entry for the `(PPC-offset, block-count)` as `(0, not-taken-count)`. The offset of zero makes the P-stream continue sequentially for `not-taken-count` instructions.

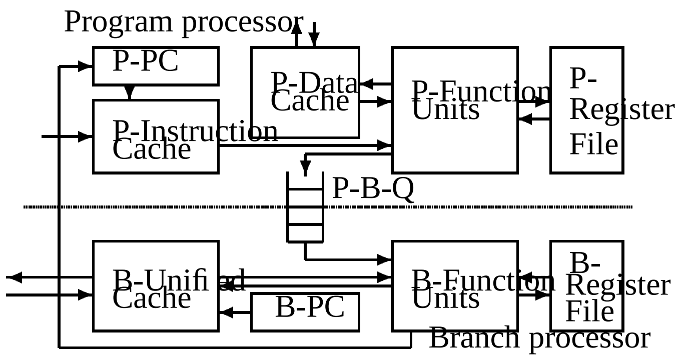
The PPC grabs a `(offset, count)` tuple from the PPCQ and adds the `offset` to the current value of the PPC, and loads a count register with `count`. The default behavior for the PPC is to increment PPC by one instruction (4 bytes for a 32-bit instruction architecture) and to decrement the count register by one. This continues until the count register is zero. A zero count register necessitates another dequeuing from the PPCQ. For more details of this architecture, the reader is referred to [10].

4 Dynamic Branch Decoupled Architecture

4.1 Architectural Organization

The dynamic branch decoupling makes the underlying decoupled streams transparent to the compiler/programmer. This clearly limits the amount of decoupling that can be performed since the hardware can only look at a limited number of instructions at a time.

The dynamic branch decoupled (DBD) processor has logically separate branch unit (BU) and program unit (PU) just as a statically decoupled processor. However, now there is a third major unit for decoupling – fetch and decouple unit (FDU). The FDU fetches instructions and decouples them into the two streams, B-stream and P-stream dynamically. The dependence resolution for the decoupling is “piggybacked” on the comparators needed for the issue



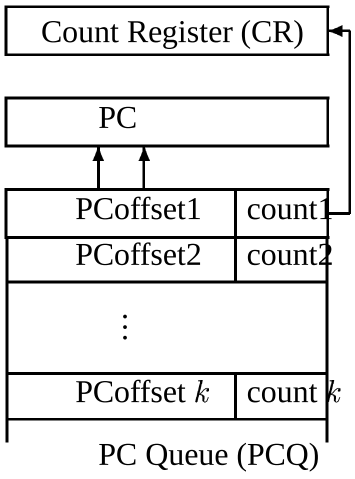


Figure 1. (a) Branch Decoupled Architecture (b) Block Diagram for P-Stream Program Counter

logic of a superscalar processor. The operations of both PU and BU are asynchronous, even though they communicate through a data-forward queue (DFQ) and share the same load-store queue (LSQ). The DFQ is used to forward some results from the BU to the PU. Certain types of instruction sequences are not decoupled. For instance, all the system calls are handled on the PU in their entirety. During those times, the BU is suspended, or put to "sleep". This functionality is also needed to handle certain dependences correctly.

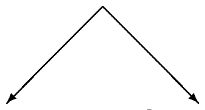
Both BU and PU have their own sets of registers, B-RF and P-RF respectively. The complexity of the underlying processors BP and PP can be different. For instance, they both could be implemented as 4-way superscalar processors. Or, the branch processor could be simpler than the program processor under the assumption that the expected load on the BU would be lower than on the PU. The experimental results answer these questions partially in Section 5.

4.2 Decoupling Logic

The fetch and decouple unit (FDU) is responsible for fetching a block of instructions in to the instruction fetch queue, and for decoupling these instructions. The fetched instructions are checked for an incoming branch instruction. The branch target address is also computed at this time. If a branch instruction and its target are known to be in the instruction window, a dependence check is performed to determine the decoupling.

An instruction is a *branch-determining-instruction* (BDI) if its destination register is *live* at the branch instruction or at another BDI and if it is used by the branch instruction or another BDI. The dependence check marks all the BDIs in the instruction window (IFQ). All the BDIs are sent to the B-instruction queue (B-IQ) at BU. All the

In IFQ:
```
LD R7, 0(R2)
SUB R4, R6, R8
AND R1, R7, R3
ADD R2, R3, R4
LSR R5, R2, R1
BEQ R2, loc1
```



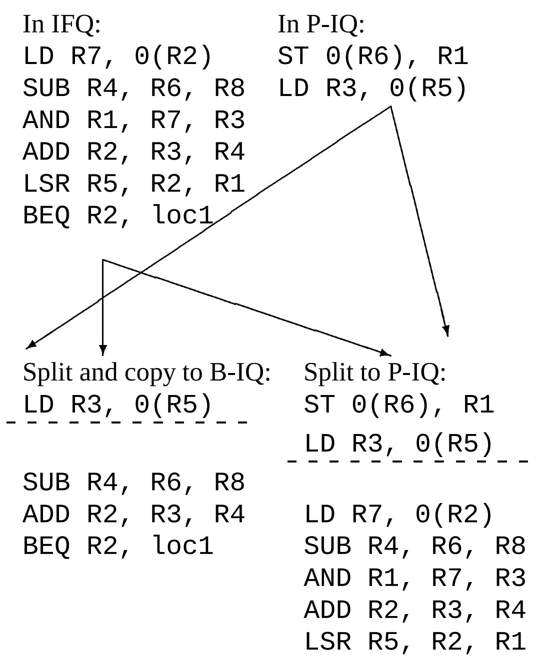
Split to B-IQ:
```
SUB R4, R6, R8
ADD R2, R3, R4
BEQ R2, loc1
```

Split to P-IQ:
```
LD R7, 0(R2)
SUB R4, R6, R8
AND R1, R7, R3
ADD R2, R3, R4
LSR R5, R2, R1
```

Figure 2. Example of Instructions Decoupling by FDU

instructions are shipped to the P-instruction queue (P-IQ). Note that some of the original instructions may have live results only within the B-stream. However, we choose to execute all the instructions at the PU to simplify the implementation.

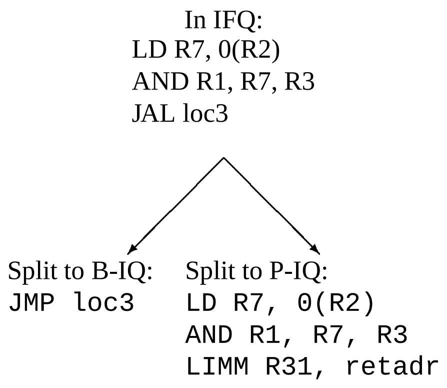
Figure 2 illustrates the decoupling with a small example program. The instructions in this example have the opcode followed by the destination register, followed by the source operands. Only SUB and ADD are found to be the branch-determining instructions in this basic block, and hence are copied to the B-stream along with the branch. Had we not copied these BDIs, ADD and SUB, to the P-stream as well,



```
In IFQ:               In P-IQ:
LD R7, 0(R2)          ST 0(R6), R1
SUB R4, R6, R8        LD R3, 0(R5)
AND R1, R7, R3
ADD R2, R3, R4
LSR R5, R2, R1
BEQ R2, loc1

Split and copy to B-IQ:   Split to P-IQ:
LD R3, 0(R5)              ST 0(R6), R1
                          LD R3, 0(R5)
SUB R4, R6, R8
ADD R2, R3, R4            LD R7, 0(R2)
BEQ R2, loc1              SUB R4, R6, R8
                          AND R1, R7, R3
                          ADD R2, R3, R4
                          LSR R5, R2, R1
```

Figure 3. Example of Instructions Decoupling with Instructions from P-IQ



```
             In IFQ:
           LD R7, 0(R2)
           AND R1, R7, R3
           JAL loc3

Split to B-IQ:   Split to P-IQ:
JMP loc3         LD R7, 0(R2)
                 AND R1, R7, R3
                 LIMM R31, retadr
```

Figure 4. Example of Decoupling Linking Jump Instruction

we would need to check if their destination registers are live in the following basic block P-stream. If yes, these values would have to be transferred to the PU's register file. In the first set of experiments, we have decided to simplify the decoupling logic by keeping the entire original basic block in the P-stream. However, we have provided a data-forwarding queue (DFQ) to forward the results computed in the BU to PU. A forwarded result can alleviate the pressure on the reservation stations and FUs in the PU.

The decoupling logic should also check for dependences of the BDIs with respect to the previously decoupled instructions sitting in the P-IQ. This is necessary because BU does not execute every instruction in the original instruction stream in IFQ. Figure 3 shows how the dependence check is extended to the instructions in P-IQ for source registers of the current basic block, in this case, `R3`, `R6` and `R8`. Note that dependence of `R4` in `ADD` instruction is "shadowed" (redefined) by the `SUB` instruction. The `LD R3, 0(R5)` instruction of previous basic block in P-IQ is copied to B-IQ since `R3` in `ADD` instruction depends on the earlier write to `R3`. This is necessary to ensure that the correct R3 value is available for `ADD` instruction. Otherwise, BU has to stall (put to sleep) till PU computes `R3` and forwards it.

There are some dependences that are not obvious. A long word based dependence is such an example. An instruction with a long word result, `ADD.L R2, R6, R4`, modifies both `R2` and `R3`. However, that information is not available in the operand in MIPS like architectures. Hence, the opcode would also have to be consulted in order to get a complete picture of the dependences. Section 4.3 presents some other examples of non-obvious dependences.

4.3 Issues and Solutions

The major shortcoming of the dynamic branch decoupling approach is that is is able to look at a very small instruction window at a time. Such a "peephole" into a program does not provide sufficient information to decouple safely in many instances. Of course, the big advantage of the dynamic decoupling approach, the availability of the run-time information more than offsets for the shortcomings. We enumerate a few limiting scenarios for dynamic decoupling, most arising out of limited context.

Procedure Calls:

On a procedure call, the stack pointer and/or frame pointer based dependences need to be considered. However, these dependences are not explicit unlike some other data dependences. In particular, consider a linking jump (or branch) which is typically used for a procedure call. It implicitly saves the return address in a fixed register ($31 in MIPS architecture, for instance). Any instruction using the return address register is dependent on the linking jump implicitly. For instance, an instruction to push the return address on the stack is dependent on the linking jump instruction. The decoupling logic has to be "aware" of these implicit dependences. Figure 4 presents an example.

The procedure calls generate other headaches for the the decoupling logic as well. Recall that the dynamic decoupling copies every instruction except a branch/jump to the program stream, and the branch/jump along with the branch-determining instructions to the branch stream. If we follow this model for the linking jump instructions, the decoupled program behaves incorrectly for the following reason. The program processor is keeping the complete program state at all times. A part of this program state is the contents of the return address register which is implicitly modified by the linking jump instruction. If the linking instruction gets moved to the branch stream without any left-over proxy in the program stream, the return address register is never modified as was intended. Hence, the dynamic decoupling inserts a proxy instruction in the program stream to load the return address (which is inserted as an immediate value in the instruction at the dispatch stage) into the return address register.

Also note that none of the stack manipulating instructions can be moved into the branch stream. This is because in a dynamic decoupling scheme, we have no control on the memory mapping. The entire dynamic decoupling is transparent to the compiler/program. The compiler makes allowance for only one program stack space. We cannot dynamically allocate another part of the memory to grow another independent stack for the branch program. This is also one reason limiting the decoupling between the program and branch streams to one block. Note that the branch stream is allowed to access data on the stack (specially if it is relative to the frame-pointer, or if synchronization points guarantee certain number of push/pop operations from the procedure entry point). However, it cannot issue any stack modifying instructions such as a push or a pop. The program stream has a write lock on the stack.

Load after Store (LAS):
A program can also have memory dependences. A store queue would usually resolve these dependences dynamically, letting a load bypass a store when there is no aliasing. However, at the decoupling stage, the aliasing has not been resolved. Hence, a load following a store cannot be decoupled into the branch stream without creating a data hazard. One option is to suspend the branch processor in such a situation and wake it again after the *store* execution. This is one kind of synchronization mechanism that we have implemented. Recall that we do not replicate *store* instructions into the branch stream since all the program state is written by the program processor.

System Calls:
The system calls have many more implicit dependences than a procedure call. It is not an explicit control altering instruction either. Hence, a safe mechanism is to let the program processor handle all the system calls and exceptions. This is what we have implemented currently. Given an extremely low frequency of system calls in the typical programs, this decision has not affected us negatively in a significant way.

5 Experimental Evaluation

In order to obtain quantitative measurement of the performance, the proposed DBD architecture is modeled and simulated using SimpleScalar toolset (version 2.0) [2]. For performance evaluation, a typical dynamically scheduled four-way superscalar five-stage pipelined RISC processor with speculative execution is used as control (base). Simulator for the base architecture is the "out-of-order" simulator in [2]. The architectural parameters for base architecture used in simulation are as follows (mostly default values): 16K level-1 instruction cache and 16K level-1 data cache; 256K level-2 unified cache; 64-entry instruction TLB and 128-entry data TLB; Memory bus bandwidth of 8 bytes with access latency of 18 cycles for 1st chunk and 2 cycles for remaining chunks; 2-level adaptive branch predictor with single 8-bit history shift register at 1st level and 1K 2-bit counter entries at 2nd level & 8K 4-way associative BTB of 512 sets; branch misprediction (fetch) penalty of 3 cycles; 5-stage pipelines: fetch, dispatch/decode, issue, writeback, commit; Fetch bandwidth of 4 instructions per cycle; Issue bandwidth of 4 instructions per cycle; RUU size 16; 4 integer ALUs, 1 integer multiplier/divisor, 4 floating point adders, 1 floating point multiplier/divisor. The performance of the base architecture simulator ("out-of-order" simulator) simulated with six of the SPEC95 benchmarks are tabulated in Table 1. The "Int Aver" is the average of "go", "li" and "m88ksim" integer benchmarks, whereas the "FP Aver" is the average of "swim", "applu" and "fpppp" floating point benchmarks. The "Average" is the overall average for the six benchmarks.

Table 1. Simulation of Base ("Out-of-order") Architecture

SPEC95	Sim Cycles [million]	CPI	IPC	BPred Accuracy
go 2stone9	663.522	1.2105	0.8261	0.7189
li boyer	115.472	0.6638	1.5066	0.9054
m88ksim test	362.095	0.7316	1.3668	0.9117
swim train	683.371	0.8579	1.1656	0.9651
applu train	361.451	0.6795	1.4716	0.9049
fpppp train	459.007	1.3916	0.7186	0.9033
Int Aver	380.363	0.8686	1.2332	0.8453
FP Aver	501.276	0.9763	1.1186	0.9244
Average	440.820	0.9225	1.1759	0.8849

5.1 DBD Parameters Exploration

We performed many simulations to find an optimal set of parameters for the DBD architecture. We start with a "default" architectural parameters for DBD architecture, and observe the impact of these parameters on the performance. Basically, we want to identify the critical path, and adjust the parameters for cost-effective performance. In addition to the parameters of base architecture, the "default" archi-

Table 2. Simulation of "Default" DBD Architecture

SPEC95	Sim Cycles [million]	CPI	IPC	Speedup
go 2stone9	523.294	0.9547	1.0475	1.2680
li boyer	81.864	0.4706	2.1251	1.4105
m88ksim test	322.628	0.6519	1.5340	1.1223
swim train	610.149	0.7660	1.3055	1.1200
applu train	345.269	0.6491	1.5405	1.0469
fpppp train	451.526	1.3689	0.7305	1.0166
Int Aver	309.262	0.6924	1.5689	1.2669
FP Aver	468.981	0.9280	1.1922	1.0611
Average	389.121	0.8102	1.3805	1.1640

tectural parameters for DBD architecture used for simulations throughout the rest of this section are as follows, unless stated otherwise: IFQ (fetch queue), P-IQ, and B-IQ length of 16 instructions, DFQ length of 16 data in word (32 bits), LSQ length of 16 outstanding accesses, PU and BU configured as the base processor.

The performance of the "default" DBD architecture simulated with the above architectural parameters are tabulated in Table 2. Speedups greater than 1 indicate performance improvement, even though different benchmarks demonstrate different degrees of speedups.

Next, we look at how the lengths of IFQ, P-IQ and B-IQ affect the performance. The purpose is to find out the critical path of DBD architecture. Besides simulating with the default queue size of sixteen instructions, IFQ, P-IQ and B-IQ are also simulated with various combinations of sizes ranging from eight to thirty-two. In addition to speedups, average queue lengths per cycle for IFQ, P-IQ and B-IQ and maximum length filled or B-IQ are also captured.

Even though queue size of B-IQ is doubled from sixteen to thirty-two instructions, none of the benchmarks show any performance improvement. This indicates that queue size of B-IQ is not part of the critical path for the "default" parameters of DBD architecture. We also observe that the P-IQ queue size may be the bottleneck as indicated by the average P-IQ length of 15.47 in "applu" simulation, i.e. P-IQ is nearly full throughout the simulation. However, only the "swim" benchmark also has close-to-full average P-IQ length, whereas other benchmarks show very low average queue length for P-IQ, between 1.3 and 2.5. Hence, this means P-IQ is only critical to "applu" and "swim" benchmarks, but may not be so for the other benchmarks. Next, in order to verify if P-IQ is the bottleneck for speedups, queue size of P-IQ is doubled from sixteen to thirty-two instructions, and B-IQ is reverted back to sixteen instruction size. However, only slight improvements are shown on "swim" and "fpppp" benchmarks Again, the P-IQ in "applu" and "swim" benchmarks are almost fully filled, 31.31 and 26.81 respectively. This indicates that the pipeline stages following P-IQ in PU are the bottleneck, which are unable to keep up with the filled P-IQ. Note that such conclusion can only be made when the preceding pipeline stages of P-IQ, which is IFQ, is held constant. This is because increase in "producer" throughput can potentially be also a factor that fills up P-IQ.

We also study the influence of DFQ on the performance of the DBD architecture. Simulations are performed with DFQ queue size of sixteen and zero. For simulation results, the fractions of data used from DFQ and data skipped on DFQ over instructions executed by PU[1] are listed. In addition, the fraction of data forwarded to DFQ over instructions executed by BU[2] and the maximum length filled for DFQ are also captured. They show the utilization of DFQ with respect to the instructions executed by BU and PU. For the simulation results of DFQ with queue size of sixteen, the fraction of data used from DFQ ("DFQ used / P-IQ") indicates the degree of contribution by data from DFQ to the performance of PU, which is closely related to overall performance since PU is on the critical path. As observed, the number of data used from DFQ for integer benchmarks is much higher than those for floating point benchmarks, ≈ 0.15 vs ≈ 0.05. This is explained by the high frequencies of integer benchmarks and their high branch dependency because BDIs are the instructions that fill the DFQ. However, the fraction of data forwarded to DFQ by BU ("DFQ filled / B-IQ") is the opposite, i.e. those for integer benchmarks are lower than those for floating benchmarks, ≈ 0.15 vs ≈ 0.40. Nevertheless, they do not contradict each other. There are two possible reasons fro these low fractions. There are fewer BDIs for branch instructions in integer benchmarks, i.e. average BDIs per branch is lower. Another possible reason is that the decoupling of integer benchmarks results in more instructions copied from P-IQ to B-IQ.

Finally, we also considered the impact of the absence of floating point units on the performance. There was no performance degradation in integer benchmarks by removing FP units from BU. For the FP benchmarks however the speedup declined from 1.0611 to 1.0524 with this change. It appears to be a minimal performance impact justifying the area savings of a "lean" BU implementation.

The summary choice for the DBD parameters depends on the types of applications. For the integer applications, a good choice is IFQ, P-IQ, B-IQ and DFQ queue sizes of 8 instructions with no floating point units supported by BU. For the FP applications, the default configuration of IFQ,

[1]Total number of instructions in P-IQ = Data used from DFQ + Data skipped on DFQ + non-BDIs.

[2]Total number of instructions in B-IQ = Data forwarded to DFQ + Instructions copied from P-IQ + Branch instructions.

P-IQ, B-IQ and DFQ queue sizes of 16 instructions with FP units in BU performs better.

5.2 Comparison with Branch Prediction

One of the goals of the DBD architecture is to achieve better overall performance compared to branch prediction. Hence, comparing simulation results between both techniques offers a perspective on how well DBD architecture performs over branch prediction. Both "optimal" configurations of DBD processor for integer benchmarks and floating point benchmarks described in the previous section are simulated. Note that neither of the processors, PU or BU, in the DBD architecture uses any kind of branch prediction. The simulation results for both configurations of the DBD architectures are shown in Tables 3 and 4 on Page 7. The configuration for processor with branch prediction is the same as the base architecture described earlier, same as that used in Table 1. However, a different set of results are captured, as shown in Table 5.

Since BU and PU in DBD architecture contain the same resources as the base architecture, it can be argued that the speedup may be due to the extra resources. Therefore, we also look at performance of enhanced base architecture which is given an equal amount of resources, shown in Table 6. To reflect the similar amount of resources found in DBD architecture for fair comparison, the data path of base architecture is doubled, hereafter referred to as "enhanced base architecture". Specifically: Fetch queue length of 16 instructions; Decode, Issue and Commit bandwidths of 8 instructions per cycle; 8 integer ALUs, 2 integer multiplier/divisor, 8 FP adders, 2 FP multiplier/divisor; and 3 memory ports. However, a careful study reveals that the enhanced base architecture may have more resources than DBD architecture. Measuring in terms of transistor count[3], the queues and decoupling logic in DBD architecture are approximately "comparable" to the 2-level branch predictor (history table + update logic). Hence, enhanced base architecture actually has a resource advantage with its 8 kbytes of BTB[4].

For simulation results, speedup shows the relative improvement of performance. CPI and IPC tell how close the performance is to the ideal. For both DBD simulations, fractions of integer branches and all branches over total instructions are also tabulated. They represent the addition of expected speedups. They are used to provide comparison between expected speedup and the actual speedup obtained. Note that these are not simulator-specific figures — they are run-time benchmark-specific statistics. Since in DBD architecture, PU executes all instructions except branches (PU executes branches within system calls and for integer-configured DBD case, PU also executes floating-point branches), the additional speedup for DBD over base architecture is expected to be at least the same as the fraction of branch instructions "saved" on P-IQ.

[3]Can't compare them in terms of functional parameters because they are different functional blocks.

[4]As well as the address decoding logic within BTB.

Table 3. Simulation of DBD with "integer" configurations

SPEC95	fraction branch	Speedup	CPI	IPC	PU idle cycles
go 2stone9	0.1464	1.2651	0.9569	1.0451	0.0396
li boyer	0.2274	1.4107	0.4705	2.1253	0.1111
m88ksim test	0.2303	1.1220	0.6521	1.5335	0.1149
swim train	0.0528	1.0898	0.7872	1.2702	0.0170
applu train	0.0336	1.0466	0.6493	1.5402	0.0001
fpppp train	0.0107	1.0087	1.3796	0.7249	0.0016
Int Aver	0.2014	1.2659	0.6932	1.5680	0.0885
FP Aver	0.0324	1.0484	0.9387	1.1784	0.0062
Average	0.1169	1.1571	0.8159	1.3732	0.0474

Table 4. Simulation of DBD with "floating point" configurations

SPEC95	fraction branches	Speedup	CPI	IPC	PU idle cycles
go 2stone9	0.1464	1.2680	0.9547	1.0475	0.0396
li boyer	0.2274	1.4105	0.4706	2.1251	0.1107
m88ksim test	0.2303	1.1223	0.6519	1.5340	0.1147
swim train	0.0635	1.1200	0.7660	1.3055	0.0179
applu train	0.0336	1.0469	0.6491	1.5405	0.0001
fpppp train	0.0137	1.0166	1.3689	0.7305	0.0018
Int Aver	0.2014	1.2669	0.6924	1.5689	0.0884
FP Aver	0.0369	1.0611	0.9280	1.1922	0.0066
Average	0.1192	1.1640	0.8102	1.3805	0.0475

To further compare performance between DBD and base architecture, we also look at the cycles wasted due to branch instructions. Note, however, that the clock period for the two architectures may differ. For DBD architecture, the fraction of idle cycles in PU due to waiting for BU to resolve branches is used, whereas for base architecture, the fraction of cycles spent in recovering from the mis-prediction is used. Both capture the branch penalty seen by the processor. Comparing the simulations of DBD processor with the integer-benchmark configuration in Table 3 and the base processor in Table 5, the three integer benchmarks demonstrate large gains in speedup, led by "li" benchmark with 1.41, "go" with 1.27 and "m88ksim" with 1.12. In fact, these gains are pretty much expected because these integer benchmarks have high percentage of non-floating point

Table 5. Simulation of Default Base Architecture

SPEC95	Speedup	CPI	IPC	mispred cycles
go 2stone9	1.0000	1.2105	0.8261	0.1678
li boyer	1.0000	0.6638	1.5066	0.1791
m88ksim test	1.0000	0.7316	1.3668	0.0972
swim train	1.0000	0.8579	1.1656	0.0419
applu train	1.0000	0.6795	1.4716	0.0103
fpppp train	1.0000	1.3916	0.7186	0.0048
Int Aver	1.0000	0.8686	1.2332	0.1480
FP Aver	1.0000	0.9763	1.1186	0.0190
Average	1.0000	0.9225	1.1759	0.0835

Table 6. Simulation of Base Architecture with Double Datapath

SPEC95	Speedup	CPI	IPC	mispred cycles
go 2stone9	1.0748	1.1263	0.8879	0.1938
li boyer	1.0010	0.6631	1.5081	0.2275
m88ksim test	1.0708	0.6832	1.4636	0.1196
swim train	1.0440	0.8218	1.2168	0.0497
applu train	1.0523	0.6458	1.5485	0.0101
fpppp train	1.1504	1.2096	0.8267	0.0057
Int Aver	1.0489	0.8242	1.2865	0.1803
FP Aver	1.0822	0.8924	1.1973	0.0218
Average	1.0656	0.8583	1.2419	0.1011

branches, 0.14, 0.23, and 0.23 for “go”, “li” and “m88ksim” benchmarks respectively. Therefore, PU is relieved from resolving these branches. The differences between branch frequencies and additional speedups are due to factors related to these branches, such as memory latency of fetching taken branches, block sizes between branches, etc.

Note that the idle cycles in PU in waiting for branches to be resolved by BU are relatively higher for integer benchmarks than for floating point benchmarks. This may sound puzzling since integer benchmarks demonstrate higher speedup. Majority of the idle cycles in PU in waiting for BU are due to back-to-back branches, i.e. block size of one. This only explains why the number of idle cycles in PU in waiting for BU is high, but it does not clarify the disparity between speedups and PU idle cycles. The probable reason for such low number of idle cycles in PU in waiting for BU is because other “idle” cycles are dominated by the bottleneck on PU in executing its instructions. These bottlenecks at execution make the idle cycles in PU in waiting for BU less critical to overall performance of floating point benchmarks.

For the simulations of DBD processor with the configuration for floating point benchmarks shown in Table 4, performance of integer benchmarks is almost identical to that in Table 3. As the configuration is intended for floating point benchmarks, a significant performance gain is observed on the floating point benchmarks. However, “applu” benchmark remains the “anomaly” among floating point benchmarks, as discussed in the previous section — because “applu” has virtually no floating point branches even though “applu” contains floating point instructions. Similar to the integer benchmarks, branch frequencies of these benchmarks represent the expected speedups because BU relieves PU from executing these branches. The difference between the actual speedup and the expected one is due to the branch penalty and other branch related factors.

Since DBD processor contains two processing units that are equivalent to one processor of base architecture, some may argue that this is not a “fair” comparison for speedup with the redundancy available in DBD processor. In order to investigate the difference in performance between DBD processor and base processor that is scaled to the same resources, the data path of base processor is doubled, including functional units and bandwidths between pipeline stages. Nevertheless, as mentioned earlier, the enhanced base processor actually has a resource advantage equivalent to its BTB. The simulation results are tabulated in Table 6, and now they are “fairly” compared with those in Table 4, as both architectures have approximately equal resources in terms of functional blocks and bus widths.

Floating point benchmarks in the “enhanced” base processor have higher speedups than their counterparts in the “floating point configured” DBD processor, where “fpppp” benchmark with 1.15 speedup for “enhanced” base processor outperforms its 1.02 speedup for DBD processor. The “applu” also shows observable improvement surprisingly. Nevertheless, the “swim” benchmark still works better in DBD processor than the “enhanced” base processor. On the other hand, the speedups for integer benchmarks in the “enhanced” base processor are relatively insignificant compared to their speedups achieved in DBD processor, ≈ 1.07 vs ≈ 1.27. This proves that DBD architecture offers higher boost in performance for integer benchmarks. This is also explained by the relatively large number of idling cycles in recovering from mis-prediction in Table 6.

6 Conclusions

We propose a dynamic variant of branch decoupled architecture in this paper. Branch decoupling attempts to benefit from shallower dependences of the branches compared with the dependence depth of the other computations. The dynamic decoupling performs the dependence analysis dynamically using the same set of comparators as used by the issue logic in a wide-issue superscalar processor. The advantage of DBD (dynamic branch decoupled) architecture over a superscalar with branch prediction arises from the explicit focus on branch dependences. A static branch decoupled architecture is also able to ‘virtually’ expand the issue window size through static program analysis.

We presented the architecture description. The DBD architecture was also evaluated through simulations with 3 SPEC95 int and 3 SPEC95 FP benchmarks. The comparison was made with a 2-level adaptive branch prediction scheme. The DBD architecture outperforms the 2-level predictor by about 16%.

The DBD architecture is really a scheduling technique with explicit priority for branches and branch determining instructions (BDIs). There are other scheduling techniques for this purpose. The future extensions of this work include the following. Currently, all the instructions are executed in the program unit processor (PP). This simplifies the synchronization structure. The register file syncs need be done only at the basic (branch) block boundaries. However, some of these instructions need not be executed in the program stream giving rise to some parallelism. But, then, we need to support synchronizations on individual registers, just as we do in static decoupling [13]. Another interesting extension would be to combine the compiler assisted static decoupling with DBD.

References

[1] P. Bird, “Data Dependencies in Decoupled Pipelined Loops,” *Interaction of Compilation Technology and Computer Systems,* edited by D. Lilja and P. Bird, pp. 87-118, Kluwer Academic, 1994.

[2] D. Burger and T. M. Austin, The SimpleScalar Tool Set, ver 2.0, Computer Science Department, University of Wisconsin-Madison, June 1997.

[3] P. Y. Chang, E. Hao, T. Y. Yeh, and Y. N. Patt, “Branch Classification: A New Mechanism for Improving Branch Predictor Performance,” Int. Symposium on Microarchitecture, Nov. 1994.

[4] M. Evers, P. Chang and Y. N. Patt, ”Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches,” Proc. 23rd Int’l Symp. on Computer Architecture, pp. 3-11, 1996.

[5] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, “PIPE: A VLSI Decoupled Architecture,” Int. Symp. on Computer Architecture, pp. 20-27, 1985.

[6] L. Kurian, P.T. Hulina, L. D. Coraor, “Memory Latency Effects in Decoupled Architectures,” IEEE Trans. on Computers, Oct. 1994.

[7] S. T. Pan, K. So and J. T. Rahmeh, ”Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation,” Proc. 5th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 76-84, 1992.

[8] J. E. Smith, S. Weiss and N. Y. Pang, ”A Simulation Study of Decoupled Architecture Computers,” IEEE Trans. on Computers, pp. 692-702, Aug. 1986.

[9] J. E. Smith, ”Decoupled Access/Execute Computer Architectures,” ACM Trans. on Computer Systems, pp. 289-308, 1984.

[10] A, Tyagi, “Branch Decoupled Architectures,” Proc. of Workshop on Interaction between Compilers and Computer Architectures at 3rd Int’l Symp. on High-Performance Computer Architecture, 1997, A summary appears in IEEE TC on Computer Architecture Newsletter, pp. 13-15, June 1997.

[11] A. K. Uht, V. Sindagi, and S. Somanathan, ”Branch Effect Reduction Techniques,” Computer, pp. 71-81, May 1997.

[12] T. Y. Yeh and Y. N. Patt, ”Two-level Adaptive Training Branch Prediction,” Proc. 24th Int’l Symp. on Microarchitecture, pp. 51-61, 1991.

[13] L. Zhang, ”A preliminary evaluation of compiler-assisted branch decoupled architecture,” MS Thesis, May 1999, Dept. of Computer Science, Iowa State University, Ames, IA 50011.