International Journal of Electronics Engineering, 1(2), 2009, pp. 209-213

Modified Architectural Support for Predicate Execution of Instruction Level Parallelism

Sweta[1], Dr. Ranjit Biswas[2] & Prof. J. B. Singh[3]

[1]Research Scholar, Shobhit University
[2]Prof. (CSE/IT) ITM, Gurgaon
[3]Prof. (CSE/IT) Shobhit University

Abstract: *Utilizing speculative execution alone to extract instruction level parallelism in the presence of branches has performance limitation .The fundamental limitation is that speculation eliminates dependencies between instructions and branches, but does not remove the branches themselves. To overcome this drawback, predicate execution is investigated. Predicate or guarded execution enables a compiler to eliminate branches from the instruction stream. As a result, many of the difficulties introduced by branches can be eliminated. This paper addresses the architectural support required to accomplish predicate execution. The architectural extensions required to provide efficient support for the predicated execution are discussed.*

Keywords: *Instruction Level Parallelism, Nullification Model, Very Long Instruction Word.*

1. INTRODUCTION

Predicate execution refers to the conditional execution of the instructions based on the value of a Boolean source operand, referred to as the predicate. If the value of the predicate is true (a logic 1), the instruction is allowed to execute normally; otherwise (a logic 0), the instruction is nullified, preventing it from modifying the processor state.

```
for ( i=0; i<=100; i++)
if (a[i]<=50)
j=j+1;
else
k=k+1;
```

Figure 1: 1 Source Code Segment of if-conversion

Figure 1.1, Figure 1.2, Figure 1.3 contains an example to illustrate the concept of predicated execution [5]. For each iteration of the loop in Figure 1.1, either the value of *j* or *k* is conditionally incremented. The basic compiler transformation to exploit predicated execution is known as if-conversion [1]. If-conversion replaces conditional branches in the code with comparison instructions that define one or more predicates. Instructions control dependent on the branch are then converted to predicated instructions, utilizing the appropriate predicate value [7]. In this manner, control dependences are converted to data dependences. Figure 1.2 and Figure 1.3, shows the assembly code for the loop example before and after if-conversion.

Note that the variables *j* and *k* have been placed in register *r*5 and *r*6, respectively. The first conditional branch *bgt* in Figure 1.2 is replaced by a predicate define instruction, *pgt*, in Figure 1.3.

```
mov     r1 ,0
mov     r2,0
id_i    r3 , a,0
L1:
Id_i    r4, r3,r2
bgt     r4 , 50, L2
add     r5,r5,1
Jump     L3
L2:
add     r6,r6,1
L3:
add     r1,r1,1
add     r2,r2,4
bit     r1,100, L1
```

Figure 1.2: Assembly Code Segment of if-conversion

```
mov     r1,0
mov     r2,0
Id_i     r3,A,0
L1:
Id_i    r4,r3,r2
pgt     p1(U),p2(U),r4,50
add     r5,r5,1  (p2)
add     r6,r6,1  (p1)
add     r1,r1,1
add     r2,r2,4
bit     r1,100,L1
```

Figure 1.3: Assembly Code Segment After if-conversion

The predicate *p*1 is assigned the value 1 if r4 > 50 and 0 otherwise. The predicate *p*2 is assigned the complement of *p*1. The instructions incrementing the value of *r*5 and *r*6 are converted to predicated instructions, associated with predicates *p*1 and *p*2 respectively. For each loop iteration, either *r*5 or *r*6 will be incremented by the predicated add instructions, contingent on the results of the predicate define instruction. The jump instruction becomes unnecessary after if-conversion.

1.1. Predicated Execution Support in Cydra 5

The Cydra 5 system is a VLIW [9], multiprocessor system utilizing a directed dataflow architecture. Each Cydra 5 instruction word contains seven operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located within the predicate register file. The predicate register file is an array of 128 Boolean (one bit) registers. Within the processor pipeline after the operand fetch stage, the predicate specified by each operation is examined. If the content of the predicate register is one, the instruction is allowed to proceed to the execution stage: Otherwise, it is squashed. Essentially, operation whose predicates are *zero* are converted to *no-ops* prior to entering the execution stage pipeline. The predicate specified by an operation must thus be known by the time the operation leaves the operand fetch stage.

```
        mov         r1,0
        mov         r2,0
        Id_i        r3,A,0
        L1:
        Id_i        r4,r3,r2
        gt          r6,r4,50
        stuff       p1,r6
        stuff_bar   p2,r6
        add         r5,r5,1     (p2)
        add         r6,r6,1     (p1)
        add         r1,r1,1
        add         r2,r2,4
        bit         r1,100, L1
```

Figure 1.4: If-then-else Predication in Cydra 5

The content of a predicate register may only be modified by one of three operations: *stuff, stuff_bar or brtop*. The stuff operation takes as operands a destination predicate register and a Boolean value as well as an input predicate register. The Boolean value is typically produced using a comparison operation . If the input predicate register is one, the destination predicate register is assigned the Boolean value. Otherwise, destination predicate is assigned to 0. The *stuff bar* operation functions in the same manner, except the destination predicate register is set to the inverse of the Boolean value when the input predicate value is one. The *brtop* operation is used for loop control in software pipelined loops and sets the predicate controlling the next iteration by comparing the contents of a loop iteration counter to the loop bound [3].

Figure 1.4, shows the previous example after if-conversion for the Cydra 5, to set the mutually exclusive predicates for the different execution paths shown in this example requires three instructions.

First, a comparison must be performed followed by a *stuff* to set the predicate register for the true path (predicate on *p*1) and a stuff_bar to set the predicate register for the false path (predicate on *p*2). This result in a minimum dependence distance of 2 from the comparison to the first possible reference of the predicate being set.

In Cydra 5, predicate execution is integrated into the optimized execution of modulo scheduled inner loops to control the prologue, epilogue, and iteration initiation. Predicate execution also allows loops with conditional branches to be efficiently modulo scheduled.

2. ARCHITECTURAL SUPPORT FOR PREDICATE EXECUTION

The hardware must support for predicated execution, some special architectural support is also required. The hardware must allocate enough spaces to store the predicated values. Even though the existing general purpose registers can be used to store these values, there are two problems of using general purpose registers. Firstly, each predicate value requires only 1 bit. Therefore, storing the value in a typical 32-bit general purpose register can be very wasteful. Secondly, because many of the predicate values and their complementary values are used, more efficient and convenient hardware support should be used to represent the complementary values. A new predicate register file design is discussed to address these problems.

Another hardware support includes providing some new instruction to set these predicate registers. These predicate registers are used by all instructions; thus, an extension has to be added to the instruction field to specify a predicate register. Lastly, the hardware must contain some logic to nullify any side effects of the instruction if the instruction is not supposed to be executed.

(Proposed Problem)

2.1. A Micro Architecture Model

A Micro architecture is composed of the processor, instruction cache and data cache sharing a common memory data bus, and the main memory subsystem. The processor supports in-order issue to the fully pipelined functional units. Each functional unit may contain up to one of each of the following: an integer unit, a floating-point unit, and load-store unit. A realistic memory subsystem is modeled to accurately show the benefits and disadvantages of new compiler techniques and architectural support. Figure 1.5 shows the five-stage pipeline including instruction fetch (IF),

instruction decode/issue (ID), instruction execute (IE), memory access (MA) and write-back/retire (WBR) to explain the execution path of instructions .

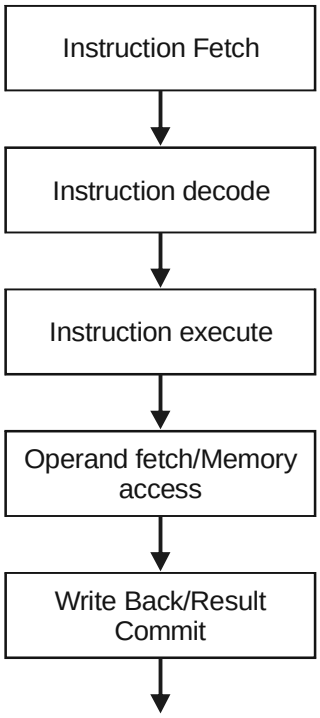

Figure1.5: Pipeline Diagram for the Micro Architecture

3. INSTRUCTION SET

Cydra 5 style of supporting full predication is chosen for the micro architecture model. Full predication offers the most efficient and flexible paradigm to support predicate execution. As a result, all instructions in the instruction set architecture are augmented with an additional source operand to hold a predicate specifier. In this manner, every instruction may be predicated. Predicate values are maintained in an N X 1 predicate register file. Predicates are manipulated via a new set of instructions are added to the baseline architecture. These instructions are classified as Predicate comparison instructions, predicate clear/set instructions and predicate save/restore instructions.

3.1. Predicate Comparison Instructions

The predicate comparison semantics [5] used and they are HPL, PlayDoh architecture. Predicate Comparison instruction compute predicate values using semantics similar to those for conventional comparison instructions. There is one predicate comparison instruction for each integer, unsigned, float, and double comparison *opcode in the* original instruction set. The major difference is that these instructions have up to two destination registers and these destination registers are in the predicate register file. The predicated comparison instruction format is shown below:

P<cmp>Pout1(<type>), Pout2(<type>), src1, src2(Pin)

This instruction assigns values to Pout1 and Pout2 according to a comparison of src1, src2 specified by <cmp>. The comparison <cmp>can be: equal (eq), not equal (ne), greater than (gt) etc. A predicate <type> is specified for each destination predicate. Predicate defining instructions are also predicated as determine by Pin.

The predicate <type> determines the value written to the destination predicate register based upon the result of the comparison and of input predicate, Pin. For each combination of comparison result and Pin, one of three actions may be performed on the destination predicate. It can write 1, write 0, or leave it unchanged, indicated by a"-". Thus, a total of $3^4 = 81$ possible type exist. There are six predicate types that are particularly effective, unconditional (U) , OR-type (OR), and AND–type (AND) predicates and their complements.

Pin	*Comparison*	*Pout*					
		$\overline{U}$	*U*	*OR*		*AND*	
0	0	0	0	-	-	-	-
0	1	0	0	-	-	-	-
1	0	0	1	-	1	0	-
1	1	1	0	1	-	-	0

Figure 1.6 truth Table for predicate types.

(Proposed Work)

3.2. Micro–architecture Extensions

To support predicate execution, some modifications to the baseline architecture has been presented. These extensions are broadly broken down into two categories: the nullification mechanism and the predicate register file.

3.2.1. Nullification Mechanism

The predicate of each instruction determines its execution state. If predicate is 1, or true, the instruction is executed normally; if the value is 0, or false the effect of the instruction are nullified. In general, nullification may be accomplished at any point in the processor pipeline before the register file or memory system is modified.

The earliest an instruction may be nullified is during the decode/issue stage. After fetching the value of an instruction's predicate, the instruction is I simply not issued if its predicate is 0. This has the advantage of the allowing the execution unit to be allocated to other operations. Thus, for critical resources such as divide units, a nullified instruction will never tie it up unnecessarily. Also, for nullified load instructions, superfluous cache and TLB misses will never be generated. On the negative side, the value of the predicate register referenced must be available during decode/issue, so the predicate register must at least

be sent in the previous cycle. This dependence distance may also be larger for deeper pipelines or if bye pass is not available for predicate registers. Increasing the dependence distance between definitions and uses of predicates may adversely affect execution time by 7 lengthening the schedule for the predicate code. This nullification model is utilized in CYDRA 5.

The other extreme for nullification is to allow the instruction to execute almost to completion , but to disallow any change of processor state in the write-back stage of the pipeline. Therefore, for instructions that write their result into the register file, this update must be suppressed. For store instructions, they must be prevented from entering the store buffer. This method is useful since it reduces the latency between an instruction that modifies the value of predicate register and a subsequent instruction which is conditioned based on that predicate register. This reduced latency enables more compact schedules to be generated for the predicated code. A drawback of this method is that regardless of whether an instruction is suppressed, it still ties up an execution unit. This method is also likely to increase the complexity of the register bypass logic and force exception signaling to be delayed until the last pipeline stage.

Hybrid nullification schemes are also possible and become more appealing for deeply pipelined machines to balance the effect of both extremes. For the current architecture, nullification at the decode/issue stage is chosen. The architecture contains a very short pipeline(five stages) and the reduced design complexity makes this the preferred choice. Also, it is believed that the negative of increased independence height incurred by this approach can be overcome with compiler transformations such as predicate promotion.

3.2.2. Predicate Register File

An N X 1 register file is used to hold predicate is added to the baseline architecture. The choice of introducing a new register file to hold predicate values rather than using the existing general purpose register file was made for several reasons. First, it is inefficient to use a 32 bit general register to hold a one bit predicate. Second, register porting is expected to be a significant problem for wide–issue processors. By keeping predicates in a separate file, additional port demands are not added to the general purpose register file, within the architecture, the predicate register file behaves no differently than a conventional register file. For example, the contents of the predicate register file must be saved during a context switch. Further more, the predicate file is partitioned into caller and callee saves section based on the chosen calling convention.

3.2.3. Predicated Execution for Out-of-order Issue Processors

Superscalar processors employing out-of-order execution via an algorithm, such as the Tomasulo algorithm, faces new problem with predicated execution. The problems mainly stem from the tagging mechanism used to forward results to instructions waiting in the reservation stations.

The Figure1.7, shows the execution of the code stream. When instruction A is issued, it deposits tag A into its destination register, again r1. Next instruction B is issued, thereby writing tag B into its destination register, again r1. Now, when the operands for instruction C are fetched, the producer of its source operand, r1, is assumed to be the last instruction to write to r1, namely instruction B. In the cases in which the predicate of instruction B is false, no result will be forwarded to instruction C, which causes an error.

```
A:    Id_i,r1,r2,r3
B:    add r1,r4,r5 (p1)
C:    Id_i r6,r1,0
```

Figure 1.7: Example of the Tagging Problem with Out-of-order.

One potential solution is not to allow instruction B to place its tag in its destination register unless its predicate is true. The problem with this is that much of the out-of-order execution capabilities of the processor are lost. With this solution, the processor must stall whenever the predicate of an instruction is available, whereas the underlying principle of out-of-order execution is to continue issuing instruction regardless if their source operands are available. The instruction not ready wait in reservation stations allowing ready instructions to bypass them. Therefore, much of the out-of-order performance potential is sacrificed with this scheme

4. CONCLUSION AND DISCUSSION

Predicate execution supports provides an effective means to completely eliminate branches from an instruction stream . Predicated or guarded execution refers to the conditional execution of an instruction based on the value of a Boolean source operand, referred as the predicate of the instruction. This architectural support allows the compiler to use an if-conversion algorithm to convert conditional branches into predicate defining instruction and instruction along alternative paths of each branch into predicated instructions. Predicated instruction are fetched regardless of their Predicated value. Instruction whose predicate value is true are executed normally. Conversely, instruction whose predicate is false are nullified, and thus are prevented from modifying the processor state. Predicated execution allows the compiler to trade instruction fetch efficiency for the capability to expose IPL to hardware along multiple execution paths.

REFERENCES

[1] V. Aho, R. Sethi and J. D. Ullman, “Compilers: Principles, Techniques, and Tools”, Addison-Wesley Publishing Company, (1986).

[2] R. P. Colwell *et al.*, "A VLIW Architecture for a Trace Scheduling Compiler", *IEEE Trans. on Computers*, (1988).

[3] Subramanian Rajagopalan, Sreeranga P. Rajan, Sharad Malik, Sandro Rigo, Guido Araujo, and Koichiro Takayama, "A Ratargetable VLIW Compiler: Framework for DSP with Instruction Level Parallelism", *IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems*, **20**, (11), (2001).

[4] Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee, "Interprocedural Probabilistic Pointer Analysis", *IEEE Transaction on Parallel and Distributed Systems*, **15**, (10), (2004).

[5] "An Architecture Framework for Introducing Predicated Execution into Embedded Microprocessors", Daniel A. Connors, David I. August, Kevin M. Crozier, and Wen-mei W. Hwuand Jean-Michel Puiatti.

[6] Anantaraman A., Seth K., Patil K., Rotenberg E., Mueller F., "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems". In: Proceedings of the International Symposium on Computer Architecture, (2003).

[7] Berg C., Engblom J., Wilhelm R., "Requirements for and Design of a Processor with Predictable Timing". In: Proceedings of the Dagstuhl Perspectives Workshop on Design of Systems with Predictable Behavior, (2004).

[8] Deverge J., Puaut I., "Safe Measurement-based WCET Estimation". In Proceedings of the Euromicro International Workshop on WCET Analysis.

[9] Fisher J. A., Faraboschi P., Young C., "Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools". Kaufmann, Los Altos, (2005).

[10] Schlansker M. *et al*, "Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity". HPL Technical Report, (1997).