Micro-Architecture Issues of Predicated Execution

Zhenghong Wang and Ruby B. Lee
Department of Electrical Engineering
Princeton University
(zhenghon, rblee@ee.princeton.edu)

Abstract- **Predicated execution appears to be a promising way to exploit more Instruction Level Parallelism. By eliminating conditional branches, branch penalties can be reduced and the size of basic blocks can be increased, further facilitating compiler optimizations. Past work on predicated execution focused almost entirely on compiler issues. In this paper, we analyze the impact of predicated execution on the pipeline control of out-of-order and in-order superscalar machines. We show problems arising in implementing predication and propose both conservative and aggressive solutions.**

I. INTRODUCTION

With technology advances, more transistors can be implemented on a microprocessor chip. To convert these extra transistors into performance, most modern high performance processors implement multiple sets of execution hardware which enable the parallel execution of multiple instructions. However, the performance gain is not proportional to the hardware cost. Due to the limited amount of Instruction Level Parallelism (ILP) that the compiler and micro-architecture can exploit, a considerable portion of hardware resources is not fully utilized. One of the major impediments to exploiting ILP is the branch instruction: it causes branch penalties, breaks a program into small basic blocks making it harder for a compiler to find parallelism, and becomes the performance bottleneck when branch resources in the processor are limited. In 1983, Allen[1] proposed a promising technique for handling conditional branches and increasing ILP called predicated execution (also called conditional execution or guarded execution). In predicated execution, the predicated instruction may be either executed or nullified depending on a guard condition called a predicate. Conditional branches can be removed via a technique called if-conversion and the codes in different paths may be merged into a single path. As a result, the reduction of conditional branch instructions reduces branch penalties, and also leads to larger basic blocks where more ILP can be found by the compiler.

Among the large number of publications on predication, most focused on compilation and instruction-set architecture issues [2-6], while some papers studied the branch behavior of predicated code [7-9]. Except for one paper [10], the micro-architecture issues in implementing predicated execution have not been covered. In this paper, we analyze the micro-architecture issues in implementing predicated execution in modern out-of-order and in-order superscalar processors. We identify problems arising in implementing predication and propose both conservative and aggressive solutions.

Section II defines the generic pipeline models of out-of-order and in-order machines. Section III discusses the micro-architecture issues of predicated execution in out-of-order machines. Section IV presents our work for in-order machines. In Section V, we draw our conclusions.

II. GENERIC MACHINE MODELS

In Fig.1 (a) and (b), we define the generic pipeline stages of out-of-order and in-order processors. In a given design, each stage of the pipeline may consist of multiple cycles.

(a) Out-of-order Pipeline Model

(b) In-order Pipeline Model

Fig.1. Generic Pipeline Models

While there are many variations, we use an out-of-order pipeline model consisting of six generic stages [11]. These are called the instruction fetch (IF) stage, the instruction decode (DE) stage, the register rename (RN) stage, the instruction Issue and register Read (IR) stage, the execute (EXE) and the register Write Back and instruction retirement (WB) stage. The functions performed in each stage will be described in section III.

In the in-order machine, we use a pipeline model composed of five generic stages [12]: the instruction fetch (IF) stage, the instruction Decode and Register read (DR) stage, the execution (EXE) stage, the memory access (MEM) and the register Write Back (WB) stage. Since register renaming is not required, the register read phase can be done at the same time as the decode phase.

We assume that the instruction set architecture (ISA) supports full predication. This is because with partial predication, only a few special instructions need a predicate, e.g., the conditional move instruction. These instructions typically have separate operation codes and can easily be handled by the hardware. On the other hand, as shown in [4], partial predication is considerably less effective than full predication, and modern advanced ISAs such as IA-64 [13] have included the full predication feature. We believe that it is important to show the impact of full predication on micro-architecture design, and that this should be carefully considered in future ISA design.

0-7803-8104-1/03/$17.00 ©2003 IEEE

Zhenghong Wang and Ruby B. Lee, "Micro-Architecture Issues of Predicated Execution," Proceedings of the Asilomar Conference on Signals, Systems, and Computers, pp.349-354, November 2003.

III. Micro-Architecture Issues in Out-of-Order Machines

To dynamically extract ILP from the incoming instruction stream, many modern high performance processors support out-of-order execution of instructions. In these processors, logical registers are typically renamed to physical registers. WAW (Write after Write) hazards and false dependencies are removed after this register renaming process. Independent instructions then can be issued and executed aggressively, not necessarily in program order.

When introducing predication into an out-of-order machine, the data dependency analysis done for a non-predicated architecture may no longer be correct, since instructions can be dynamically nullified. We give examples below. For correct execution, the micro-architecture must be redesigned. After careful analysis, we identified three sources of problems in the pipeline: register renaming, instruction issue and instruction retirement.

A. Register Renaming

Register renaming is a popular technique in modern out-of-order processors. In the register renaming stage, the destination register (logical register) of each instruction is assigned to a different physical register. The mapping relationship is then stored in a structure, e.g., a Register Alias Table (RAT). The source registers of an instruction are then renamed by looking up the RAT for the corresponding physical registers that are defined in previous instructions.

Before Register Renaming		After Register Renaming	
ld	R3 ← mem(R30)	ld	Ra ← mem(Rx)
add	R4 ← R5, R3	add	Rc ← Rd, Ra
add	R3 ← R1, R2	add	Rb ← Re, Rf
sub	R6 ← R1, R3	sub	Rg ← Re, Rb

Fig.2. Code example of register renaming

In Fig.2, a false dependence (or Write After Write hazard) exists between the first and the third instructions: their destination registers are both R3. Without register renaming, if the `ld` instruction causes a cache miss and out-of-order execution is allowed, the third instruction may get executed and write R3 before the `ld` instruction. The second instruction may thus get the wrong operand. To ensure the correctness, the instructions have to be executed serially. However, by applying register renaming, the first R3 is renamed to physical register Ra and the following use of R3 refers to Ra by looking up the RAT with R3. Then the logical register R3 in the third instruction is assigned to physical register Rb (and the corresponding entry for R3 in the RAT is updated with Rb). The next use of R3 then refers to Rb since the RAT now maps R3 to Rb. So, no matter what order the two instructions that define R3 are executed, the following instructions that use R3 will always get the correct results.

Problems in Register Renaming with Predication

Unlike the traditional execution model, instructions in the predicated execution model can be dynamically nullified, i.e., the mapping of the destination register of the instruction may be invalid. The following instructions that use this logical register thus cannot use the mapping until they know whether the defining instruction is executed nor nullified, i.e., until the predicate of the defining instruction is resolved. The following example shows the problem if the register renaming logic is not changed.

Before Register Renaming	After Register Renaming
P0 addi R3 ← R0,#5	P0 addi Ra ← R0,#5
P0 cmp.eq P1,P2 ← R1,R2	P0 cmp.eq Pa,Pb ← Rc,Rd
P1 add R3 ← R1,R2	Pa add Rb ← Rc,Rd
P0 add R5 ← R3,R4	P0 add Rf ← Rb,Re

Fig.3. Code example of register renaming with predication

In Fig.3, R3 has multiple definitions before its use in the fourth instruction. In normal register renaming procedure, the two defining instructions will rename R3 to two different physical registers, say Ra and Rb, and the fourth instruction will read Rb for its operand since the latest mapping for R3 in the RAT is Rb. The second instruction compares R1 and R2 for equality and sets the logical predicate registers P1 and P2 accordingly. (The corresponding physical predicate registers are Pa and Pb.) If P1 (or physical predicate register Pa) is set to true, the mapping R3 → Rb is valid, and this code will run correctly. However, if P1 (Pa) is evaluated as false, the third instruction is nullified and the fourth instruction should use Ra instead of Rb!

Possible Solutions

Conservative Register Renaming. The processor can simply stall the pipeline whenever an unresolved predicate is met and keep waiting until the predicate is available (Fig. 4). This solution, however, may significantly degrade performance.

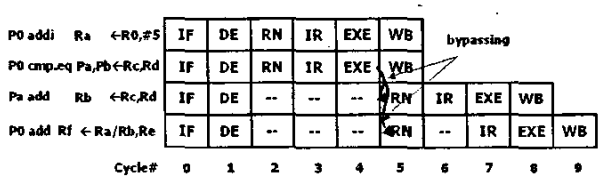

Fig.4. Performance degradation due to conservative register renaming

Due to the conservative register renaming, even with a comprehensive bypass network, the `cmp` instruction in the above code segment causes a 3-cycle penalty. This is equivalent to a 3-cycle branch penalty at 100% mispredication rate. The benefits that predication may bring by eliminating branches are now all lost. This example shows the need to reconsider the design of the register renaming unit.

Software Solutions. Before we redesign the micro-architecture of the register renaming logic, we see whether there are simple software solutions.

One compiler technique is to schedule the predicate-define instruction as early as possible, so that useful, independent instructions can be executed between it and the instruction that depends on it. This approach, however, is not always effective in eliminating all the pipeline stall cycles due to the programs' inherent flow dependency.

Another compiler techniques is called the Static Single Assignment (SSA). In SSA, different registers are assigned to the same variable in different paths. A ϕ-function is inserted afterwards to select the correct result.

$x \leftarrow \phi(x1, x2)$ is defined as follows:

P1 mov Rx ← Rx1

P2 mov Rx ← Rx2

where P1 and P2 are the predicates for the two paths, and Rx, Rx1 and Rx2 are the logical registers for variables x, x1 and x2. Fig.5 shows two examples.

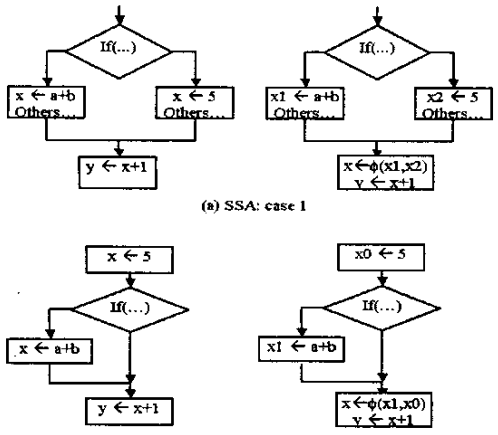

Fig.5. Static Single Assignment Examples

Since in SSA each register has only one definition, the register renaming logic can simply do the renaming without considering the predicates. But at the point where the ϕ-function must select the correct result, the predicates must be available. Again, this cannot always be achieved without pipeline stall cycles.

Hence, software techniques cannot completely solve the register renaming problem to prevent stalls in the pipeline. Furthermore, making the performance of an out-of-order machine rely heavily on compiling techniques is not desirable since an out-of-order machine is supposed to perform well even for less optimized code.

Hardware Static Single Assignment. Although software SSA assigns a different register to each variable in each path of the control flow, a ϕ-function is always needed at the merge point where the result is written to one register. Unless the predicates are available before the execution of the ϕ-function, this will still cause one or more pipeline stall cycles. However, if the hardware can implement the ϕ-function such that there are no multiple writes to a single register, the problem is solved. Based on this idea, a hardware version of SSA can be implemented as follows:

Augment the RAT into a multiple-column table, i.e., for each logical register, its corresponding entry in RAT has multiple fields which are used to store multiple possible sources of this logical register (see Fig. 6). For normal instructions, i.e., the predicates are known to be true, do the register renaming and update the RAT entry with the newly mapped physical register address. The address is put in the first field of the entry, all other fields are cleared. For nullified instructions, i.e., the predicates are known to be false, skip the register renaming and keep the RAT unchanged.

For an instruction with an unresolved predicate, a new physical register is assigned to the destination register. This mapping along with the unresolved predicate is put in the first unoccupied field of the entry of the RAT.

Whenever an entry of the RAT is full, or there is a use of a register whose entry shows multiple sources, a special "internal" instruction, the select instruction, is inserted into the decoded instruction stream. Indeed, the select instruction plays the role of the ϕ-function in software SSA. It selects the correct operand from the possible sources and generates a new register that will be used by the following instructions that need this operand. The select instruction is assigned a physical register which is used to store the selected result. The corresponding logical register is then mapped to this physical register, i.e., the first field of the RAT entry of the logical register is updated with the assigned physical register address, all other fields in this entry are cleared. This ensures that the following uses of the logical register will refer to the "output" of the select instruction which is the correct result.

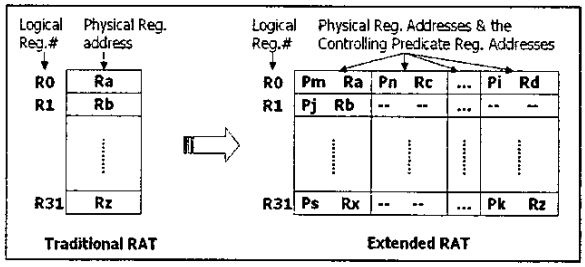

Fig.6. Hardware SSA: Extending RAT

In hardware SSA, the `select` instruction plays the role of the ϕ-function in software SSA. It waits for the results of

unresolved predicates and whenever the correct source can be determined, it selects the correct results. Since only select instructions deal with unresolved predicates, a normal instruction will always use operands from a normally executed instruction or from the `select` instruction, and thus can be renamed without ambiguity. Fig.7 shows the execution of the same code used in Fig.4. By inserting a select instruction, no stall is needed at the register renaming stage. In this example, only data dependencies cause bubbles at cycles 3 and 4: the `select` instruction must wait for the new value of Pa and the last instruction must wait for the result of the `select` instruction.

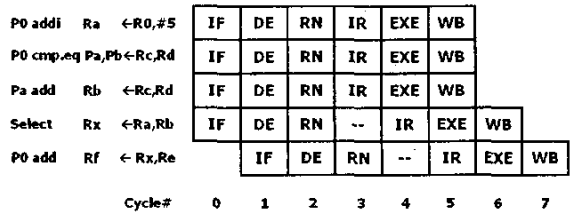

Fig.7. Code Example: Hardware SSA

Although the hardware SSA scheme is a more aggressive micro-architectural solution for the problem in register renaming for predicated code and it does not require significant modification of the traditional pipeline, it has some drawbacks. First, it requires a larger RAT which may lengthen cycle time since the RAT access is in the critical path of the register renaming stage. Second, the inserted instructions consume extra resources, e.g., reservation station entries, and thus may degrade overall performance. Third, though hardware SSA can be easily implemented on some pipelines such as the Itanium pipeline [12] where the select instruction can be implemented as a special micro-operation, it may not be compatible with other pipeline implementations where micro-operations are not used.

B. *Instruction Issue*

In a typical out-of-order machine, instructions enter a pool after they are decoded and register renamed. When all operands of an instruction in the pool are available, it is ready to execute. The wakeup/select logic chooses instructions from the ready ones and sends them to the functional units. Fig.8 shows a possible implementation.

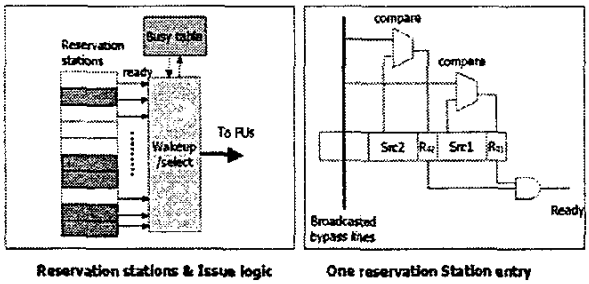

Fig.8. Reservation stations and Issue logic Architecture

In this implementation, the instruction pool is made up of the reservation station entries where the "ready" signal is asserted as true. Each entry of the reservation station has two fields that correspond to the two operands. Each field stores the physical register address it received in the register renaming stage, and has a ready bit indicating whether the operand is already available. The execution module broadcasts the newly generated result along with its corresponding physical register address, and each reservation station entry checks this register address with both of its two operand fields. If there is a match, the ready bit of the corresponding operand is set. When both ready bits are set, the instruction is ready to execute.

Problems in instruction issue with Predication

Unlike the traditional execution model, each instruction in the predicated execution model is not only dependent on the usual operands, but also dependent on its predicate. This must be considered in the instruction issue stage because the ready conditions are different.

Architectures with Predication Support

When considering the predicate, two issue policies can be adopted.

- Conservative Policy: the predicate is considered as another operand. The instruction can be issued only if both its usual operands and its predicate are ready.
- Aggressive Policy: the instruction can be issued once its usual operands are ready, regardless of the predicate. Here we utilize the fact that the correctness of the result of an instruction is not dependent on the predicate: if the predicate is true, the result is correct; if it's false, the result will be discarded whether or not it is correct, since the instruction is nullified.

To implement the above two issue policies, we need to modify the original micro-architecture slightly. Fig.9 shows the reservation entry structure for conservative issue policy.

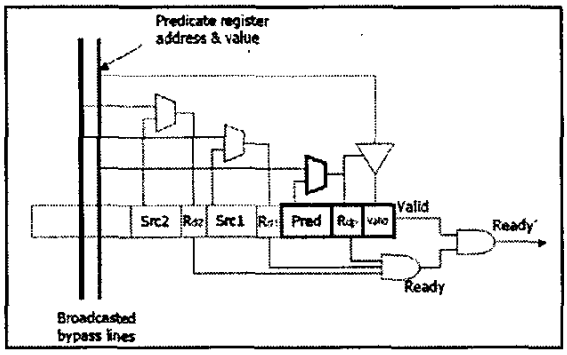

Fig.9. Reservation Station Entry: Conservative Issue Policy

The predicate is regarded as an extra operand. Thus one more operand field is added to each reservation station entry. Similar to the other two fields, it contains a physical predicate register address and a ready bit. The new ready signal will be set only when all the three ready bits are set. In addition, a valid bit is added into the reservation station entry, which indicates whether this instruction will be executed or nullified. An instruction can be issued only if it is both valid and ready (as shown by Ready' in Fig. 9). The wakeup/select logic does not need any change since it just chooses the instructions from the ready ones as before.

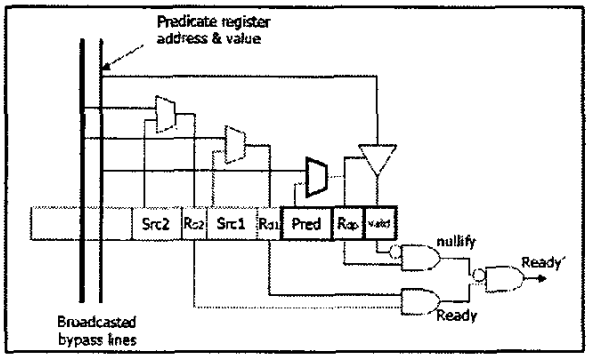

Fig.10. Reservation Station Entry: Aggressive Issue Policy

The structure of reservation station entry for the aggressive issue policy is similar in terms of the fields for operands and the predicate. However, the Ready' signal generation is different (see Fig. 10). In the aggressive policy, the two usual operands being ready indicates that the instruction is possibly ready. The predicate is checked to see whether the instruction is nullified. The policy is aggressive in that unless the instruction is known as nullified, i.e., the predicate is resolved and is evaluated as false, it is regarded as ready once the usual operands are ready.

We conclude that the micro-architecture of the instruction issue stage only needs slight modification in the structure of reservation station entries. The wakeup/select logic does not need any change.

C. *Writeback and Instruction Retirement*

In the writeback and instruction retirement stage, the processor writes results to the logical destination registers and releases physical renaming registers. The instructions can retire either in-order or out-of-order. Although out-of-order retirement may have better performance, most modern processors choose in-order retirement. The main reason is that in-order retirement maintains the precise exception property and is much simpler in micro-architecture implementation.

For a processor with predication support, instructions can also retire either in-order or out-of-order. The micro-architecture of in-order retirement is almost identical to that of the non-predicated processor. The only change may be the logic that is used to check whether an instruction still has an unresolved predicate.

Unlike non-predicated code, the precise exception property can still be maintained even if the instructions do not retire in-order. Two retire policies are possible:

- Conservative retire policy: in-order retire. This retire policy leads to simple micro-architecture as we have explained above.
- Aggressive retire policy: partial out-of-order retire. Normal instructions retire in-order, while nullified instructions can retire as early as possible.

Since nullified instructions do not change machine state, they can be retired out of program order, without affecting precise exceptions. This "partial" out-of-order retirement brings some potential benefits, such as more free physical registers and more available reservation station entries and reorder buffer entries. This may reduce pipeline stalls due to insufficient resources and thus lead to higher performance.

Fig.11 shows an in-order and partial out-of-order retire policy. For in-order retirement, the logic only needs to check a fixed number of instructions and add the released physical registers to the free register list. The partial out-of-order retirement needs to check more early-retirement instructions and requires more write ports to the free register list.

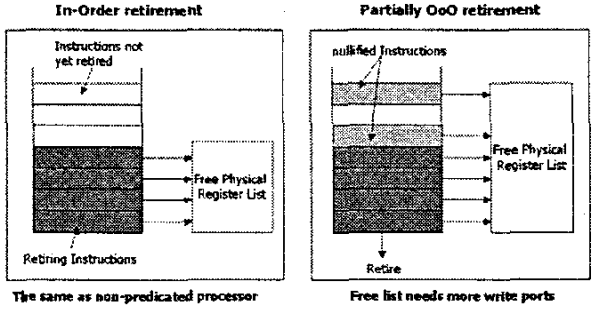

Fig.11. In-order retire vs. Partial out-of-order retirement

IV. MICRO-ARCHITECTURE ISSUES FOR IN-ORDER MACHINES

Unlike out-of-order machines, in-order machines do not need register renaming, reservation stations, wakeup/select logic and a free register list, and thus do not have the related problems discussed in Section III. The extra logic needed for predication support is just piping the predicate bits in pipeline registers and the predicate bypass network.

A. *Bypass Network Analysis*

The in-order execution ensures that the predicate-define instructions always get executed before the predicate-use

instructions. Thus the predicate is always resolved before the EXE stage of the predicate-use instruction. Fig.12 shows an example.

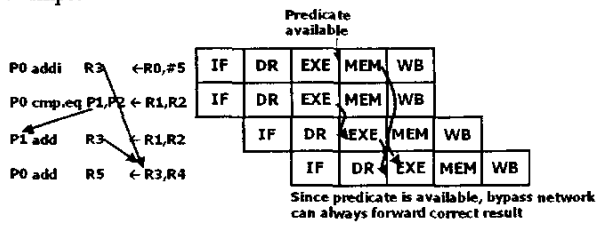

Fig.12. Code Example: Bypass Network Analysis

In this example, we assume the compiler schedules the predicate-define instruction (cmp.eq) before the predicate-use instruction (P1 add). The predicate-define instruction generates its result at the end of its EXE stage. Due to the in-order execution, the next predicate-use instruction cannot be executed in the same cycle as the first one. Thus the predicate is always available at the beginning the EXE stage of a predicate-use instruction; if the instruction is nullified, its result can be discarded correctly.

V. CONCLUSIONS

In this paper, we analyzed the micro-architecture issues in implementing predicated execution and proposed alternative solutions. For the out-of-order machine model, we show that the Register Renaming stage is most impacted by predicated execution. In a conservative solution, register renaming may significantly degrade performance due to pipeline stalls. Otherwise, significant modification to current micro-architecture may be needed to improve the performance. Current solutions are not optimal and efficient register renaming for predicated execution is still an open problem.

The Instruction Issue stage is also impacted. The complexity of the reservation station circuit is slightly increased to add the effects of the predication register and generate a new Ready' signal, based on either conservative or aggressive policies. The wakeup/select logic does not need any changes.

In the Instruction Retirement stage, an aggressive policy of retiring nullified instructions as soon as possible, in an out-of-order manner, can be achieved while still maintaining the precise exception property. Early retirement of nullified instructions may lead to higher performance by freeing up more physical registers, reservation stations and reorder buffer entries. However, more write ports of the free register list are required for this partial out-of-order retirement.

For an in-order machine, we found that predicated execution can be integrated easily into the pipeline, with very few changes.

REFERENCES

[1] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren, " Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Language*, pp.177-189, January 1983

[2] S. A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, R.A. Bringmann, and W.W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp.45-54, December 1992

[3] Nancy J. Warter, Daniel M. Lavery and Wen-mei W. Hwu, "The Benefit of Predicated Execution for Software Pipelining," *Proceedings of HICSS-26*, vol.1, pp.497-506, January 1993

[4] S.A. Mahlke, R.E. Hauk, J.E. McCormick, D.I. August, and W.W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture ISCA 22*, pp.138-149, May 1995

[5] D.I. August, W.W. Hwu, and S.A. Mahlke, "A Framework for Balancing Control Flow and Predication", in *Proceedings of the 30th International Symposium on Microarchitecture*, November 1997

[6] August, J. Sias, J. Puiatti, S. Mahlke, D. Connors, K. Crozier and W. Hwu, "The Program Decision Logic Approach to Predicated Execution", in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999

[7] D.N. Pnevmatikatos and G.S Sohi, "Guarded Execution and dynamic branch prediction in dynamic ILP processors," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp.120-129, April 1994

[8] G.S. Tyson, "The Effects Of Predicated Execution On Branch Prediction", in *Proceedings of the 27th International Symposium on Microarchitecture*, pp.196-206, November 1994

[9] G.S. Tyson and Matthew Farrens, "Evaluating the effects of predicated execution on branch predication," *International Journal of Parallel Processing*, vol.24, no.2, pp.159-186, 1996

[10] P.H. Wang, H. Wang, R.M. Kling, K. Ramakrishnan and J.P. Shen, "Register Renaming and Scheduling for Dynamic Execution of Predicated Code," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp.15-25, January 2001

[11] Palacharla, S.; Jouppi, N.P.; Smith, J.E, "Complexity-Effective Superscalar Processors," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp.206-218, June 1997

[12] John L. Hennessy and David A. Patterson, "Computer Architecture – A Quantitative Approach," Morgan Kaufmann Publishers, Inc., San Mateo, CA, second edition, 1995

[13] Intel, "IA-64 Architecture Software Developer's Manual, Vol.3: ISA Reference," Rev 1.1, July 2000