# DETECTION OF SOFTWARE DATA DEPENDENCY IN SUPERSCALAR COMPUTER ARCHITECTURE EXECUTION

**Elena ZAHARIEVA-STOYANOVA, Lorentz JÄNTSCHI**

*Technical University of Gabrovo, Bulgaria, zaharieva@tugab.bg*
*Technical University of Cluj-Napoca, Romania, lori@webmail.academicdirect.ro*

Abstract: This paper treats the problem of detection of data hazards in superscalar execution. The algorithms of independent instruction detection are represented. They can be used in out-of-order execution logic and a code optimized algorithm. The algorithms use the platform of Intel Pentium architecture and analyze the IA-32 instruction set. The implementation of the algorithms is in a software simulator, which represents the way the Intel Pentium Processor works. It can be used in software module, which simulate out-of-order execution logic.

Keywords: simulators, RISC architecture, Intel Pentium processor, IA-32 architecture, data hazards, data-flow analysis

## 1. INTRODUCTION

Instruction-level parallelism is a frequently used technique in up-do-date processors' architectures. It makes it possible to execute more than one instruction per cycle. Today's processors use more than one pipeline, which means that they have superscalar architecture.

Although the pipeline usage is a feature of RISC processors, this technique is used also in processors with mixed architecture - a mix of RISC and CISC. For example, developing IA-32 architecture, Intel Corporation introduced superscalar technique in the Pentium processor. The first Intel Pentium processor has two 5-stage pipelines. Next comes 3-ways supersclar P6 architecture with 10-stage pipelines (Keshava and Pentkovski, 1999). The number of pipeline stages in NetBurst architecture is increased to 20 (Hinton et al. 2001).

Because of the possibility to execute more than one instruction per cycle, the instruction-level parallelism increases the performance highly. On the other hand, an ideal sequence of uniform instructions is rare. The execution of one instruction often depends on the result of the previous instruction's execution. This situation is called ***data hazard***.

Data hazards make the performance lower than that of one-pipeline architectures. The situation when the next instruction depends on the results of the previous one is occurred very often. It means that these instructions cannot be executed together. For example, the first Intel Pentium has two 5-stage pipelines. If two neighbor instructions are independent they could be decoupled and executed together in U and V pipeline. If there is data dependency in the instructions, the second instruction waits to be decoupled with a next one.

To decrease the influence of data hazards, P6 processor architecture introduced the concept of ***dynamic execution.*** It makes it possible to get as far as out-of-order execution in a superscalar implementation. Dynamic execution incorporates the

concepts ***dynamic data flow analysis*** and ***speculative execution.***

Dynamic data flow analysis involves real-time analysis of the flow of data through the processor to determine data and register dependencies and to detect opportunities for out-of-order instruction execution. Speculative execution refers to the processor's ability to execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream.

This paper treats the problem of detection of data hazards in superscalar execution. The algorithms of independent instruction detection are represented. They can be used in out-of-order execution logic and a code optimized algorithm. The algorithms use the platform of Intel Pentium architecture and analyze the IA-32 instruction set. The algorithms implementation is involved in a software simulator of Intel Pentium architecture.

## 2. DATA HAZARDS IN SUPERSCALAR ARCHITECTURES

The superscalar architecture makes it possible to execute more than one instruction per cycle. To execute several instructions simultaneously, the instructions have to be arranged in an ideal sequence, and that happens rarely. Every deviation from the ideal sequence of uniform instructions is called ***a hazard*** (Hlavicka, 1999).

The hazard is situation that prevents the next instruction in the instruction stream from executing during its designated clock cycle. Hazards in pipelines can make it necessary to stall pipelines. They reduce the performance from the ideal speedup gained by pipelining and superscalar execution.

There are three types of hazards: ***structural***, ***data*** and ***control*** hazards.

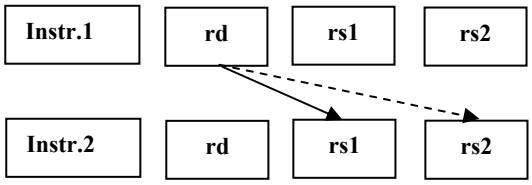
Structural hazards arise from resource conflicts in the hardware. In these cases, the hardware cannot support some combination of instructions in simultaneous overlapped execution. Data hazards appear when the execution of an instruction depends on the results of previous instruction. Control hazards arise from the pipelining of branches, calls and other instructions that change Program Counter. Control hazards reduce the performance when these types of instructions occur in a program very often.

Structural hazards could be avoided by duplicating hardware resources. To avoid control hazards, the branch prediction technique is used.

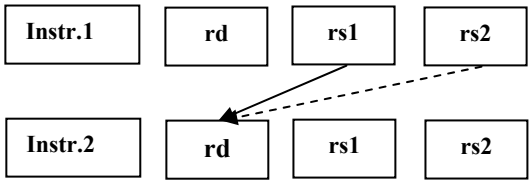
Data hazards are more common than the rest. As it was mentioned, they arise because of data interdependency in the instructions' order. Depending on the order of read and writes access in the instruction data hazards may be classified as follows:

- RAW (read after write);
- WAR (write after read);
- WAW (write after write).

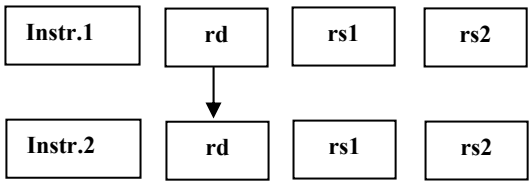
RAW data hazard is the most common type. It arises when the next instruction tries to read a source before the previous instruction writes to it. So, the next instruction gets the old value incorrectly. WAR hazard arises when the next instruction writes to a destination before the previous instruction reads it. In this case, the previous instruction gets a new value incorrectly. WAW data hazard is situation when the next instruction tries to write to a destination before a previous instruction writes to it and it results in changes done in the wrong order.




(a) RAW data hazard




(b) WAR data hazard




(c) WAW data hazard

Fig. 1. The examples of data hazards

The software dependencies between two neighbor instructions are given on fig. 1. Fig. 1a describes the RAW data hazard - the destination register of the first instruction and a source register of the second instruction are the same.

WAR data hazard could be arises if an instruction needs more than one cycle for execution. For example, if a source register of the first instruction and destination register of the second instruction is the same; and if second instruction execution is faster the first one, it is possible to arise WAR data hazard (fig 1b).

The situation described on fig. 1c. is similar - if the second instruction is faster than the first one; and if

they use the same destination; it is possible the result from the first instruction to be written after the result from second one.

It is difficult to find a solution to the problem with data hazards. One possible solution of this problem is a simple hardware technique called a ***forwarding*** or ***bypass***. This technique works as follows: The ALU result is always fed back to the ALU input latches. If the hardware detects that the next instruction uses the results from the previous instruction, the control logic selects the forwarded result as the ALU input rather than the value read from the register files.

RAW data hazards make the performance lower than that of one-pipeline architectures. The situation when the next instruction depends on the results of the previous one is occurred very often. It means that these instructions cannot be executed together. For example, the first Intel Pentium has two 5-stage pipelines. If two neighbor instructions are independent they could be decoupled and executed together in U and V pipeline. If there is data dependency in the instructions, the second instruction waits to be decoupled with a next one. (Pentium, 1998)

To decrease the influence of data hazards, P6 processor architecture introduced the concept of ***dynamic execution.*** It makes it possible to get as far as out-of-order execution in a superscalar implementation (Intel Corporation, 2000, 2001). Dynamic execution incorporates the concepts ***dynamic data flow analysis*** and s***peculative execution.***

Dynamic data flow analysis involves real-time analysis of the flow of data through the processor to determine data and register dependencies and to detect opportunities for out-of-order instruction execution. Speculative execution refers to the processor's ability to execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream.

## 3. DETECTION OF SOFTWARE DATA DEPENDENCY

To find data hazards in program execution order, it is necessary to observe neighbor instructions. If they use the same sources and a destination, it is possible to arise some type of data hazards.

The number of the neighbor instruction observed by hazard detection algorithm depends on the pipelines' number. For example, two-way superscalar architecture in Intel Pentium P5 needs to find whether there is dependency between two neighbor instructions.(Pentium, 1998)

### *3.1 Algorithm for detection of software data dependency between two instructions*

To determine software dependency between two neighbor instructions, it is necessary to detect whether the use the same data operands or not. Instructions use register, memory, or immediate for a data operand. In this paper the RISC architecture features are used.

RISC processors have Load/Store architecture. It means that only two special instructions use memory operand: Load and Store. Other instructions, like ALU type addition and subtraction, use registers and do not use memory.

RISC instructions use three operands. In this case, the source operands are reusable. The common instruction format is:

**Instruction Rd, Rs1, Rs2,**

where: Rd is destination; Rs1, Rs2 are sources.

The algorithm keeps the result data in a buffer with size equal to the number of instructions. If there is data dependency between instructions, the corresponding value in the buffer is: 1 for RAW, 2 for WAR, 4 for WAW. Otherwise, the value is 0. The value at the start is 0 because the first instruction is independent.

Each instruction could be described with following information:

- code - instruction code;
- Rd - destination;
- Rs1 - first source;
- Rs2 - second source.

The algorithm for detection of software dependency work as follows:

***Step 1: Determining of first instruction as undependable.***

buffer[0]=0;

for I = (first instruction) to (last instruction – 1)

{***Step 2: Determine the information for the instruction I: code1, Rd1, Rs11, Rs12***

***Step 3: Determine the information for the instruction I+1: code2, Rd2, Rs21, Rs22***

***Step 4: Determine if there is RAW data hazard:***

if ((Rs21==Rd1) or (Rs22==Rd1)) buffer[I]=1;

***Step 5: Determine if there is WAR data hazard:***

if ((Rs11==Rd2) or (Rs12==Rd2)) buffer[I]=2;

***Step 6: Determine if there is WAW data hazard:***

if ((Rs11==Rd2) or (Rs12==Rd2)) buffer[I]=4;

}

Using C/C++ programming language, the algorithm's representation is:

```
struct instruction
{ char code[5];
```

```
  char rd[8];
  char rs1[8];
  char rs2[8];
  unsigned char address_mode;
};

struct instruction prev, next; // prev, next keep information about two neighbors instructions

int i, n;            // n is number of instructions

unsigned char *buffer;      // the result buffer

buffer=new char [n];

buffer[0] = 0;      // first instruction is independent

for (i=1; i<n; i++)

 {

  /* determining the information for the previous instruction i-1; prev stores this information */

   /* determining the information for next instruction i; next stores this information */

 if(strcmp(prev.rd,next.rs1)||strcmp(prev.rd,next.rs2)) buffer[i] = 1;

 else buffer[i]=0;

 if(strcmp(next.rd,prev.rs1)||strcmp(next.rd,prev.rs2)) buffer[i] |=2 ;

   if(strcmp(next.rd, prev.rd) buffer[i] |=4 ;

}
```

The real program have to verify if there are identical registers: EAX and AX, for example. It is depends on the particular algorithm application. The information about instruction code and addressing mode is needless in most cases. This information is used for full instruction description.

### *3.2. Advanced algorithm for detection of software data dependency*

The algorithm for detection of data dependency gives information about dependency of two-neighbor instruction. It is more interesting to find data dependency between more than two-neighbor instructions.

In this case, the algorithm can be modified as follows: The external cycle to value of k is introduced. The value of k defines for how many instructions has to the algorithm determine is it data dependency or not. For example, if the algorithm searches data dependencies between three neighbor instructions, the value of k is 3.

Using C/C++ programming language, the algorithm's representation is:

```
int j, k;            /* k value determines the farthest instructions, which have data (most remote instructions, who have data) interdependency */

for (j=1; j<k; j++)

 { // reset first j buffer items

   for (i=1; i<n; i=i+j)

   { /* determining the information for the previous instruction i-j; prev stores this information */

     /* determining the information for the next instruction i; next stores this information */

if(strcmp(prev.rd,next.rs1)||strcmp(prev.rd,next.rs2)) buffer[i][j-1] = 1;

else buffer[i][j-1]=0;

if(strcmp(next.rd, prev.rs1)||strcmp(next.rd,prev.rs2)) buffer[i][j-1] |=2 ;

 if(strcmp(next.rd,prev.rd)buffer[i][j-1]|=4 ;
 }
}
```

It is more appropriate to treat the result data buffer as a matrix. The matrix rows define whether there is data dependency between instructions. The matrix columns define in which neighbor instruction there is data dependency. For example, first column shows dependency between two-neighbor instructions; second column determines a dependency between three-neighbor instructions and so on. If all values in a matrix row are 0, it means that the instruction is independent.

The algorithm detects data dependency between instructions in program order. It can be used for data flow analysis. The result data buffer contains information about out-of-order execution logic.

## 4. APPLICATION OF SOFTWARE DATA DEPENDENCY DETECTION ALGORITHMS

The represented algorithms are implemented in software simulator of Intel Pentium architecture.

Simulation is a frequently used technique in computer architecture development. The software simulators could be used as a tool for studying these architectures and optimisation processes. The existing demo programs show these principals just as an overall picture. The reason to create a new simulator is to show the base concepts of the Intel Pentium processors working by means of short assembler programs. This simulator could be used also for source code efficiency evaluation. (Zaharieva-Stoyanova, 2002)

The existing demo programs, for example DynExec, show these principals just as an overall picture. The reason to create a new simulator is to show the base concepts of the Intel Pentium processors working by means of short assembler programs. This simulator could be used also for source code efficiency evaluation. This type of simulator would be very useful in higher-school education to illustrate the pipelining in a superscalar architecture. Moreover, it shows a commonly used real processor as Intel Pentium.

The base structure of Intel Pentium processor simulation model of is shown on fig. 2. The Graphics User Interface (GUI) consists of the following forms:

- ***Source code window*** - it shows the source code and the currently executed instructions in the code segment (.code). The currently executed instructions are shown as well.
- ***Data window*** – it shows the contents of the memory bytes, where the standard data segment (.data) is allocated.
- ***Stack window*** – it shows the contents of the memory bytes, where the stack segment (.stack) is allocated.
- ***Registers window*** - it shows the contents of general-purpose registers (EAX, EBX, ECX, EDX, EBP, ESI, EDI) and the flags (Eflags). At the beginning the registers are cleaned.
- ***Pipelines window*** - this is a graphics representation of the work of the two pipelines for the next 5 -10 cycles.

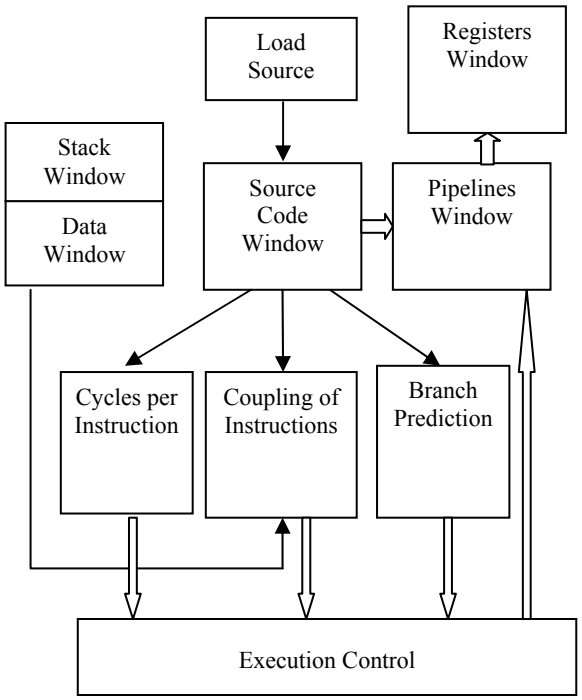



Fig. 2. The structure of Intel Pentium architecture software simulator

Apart from the GUI, the program includes the following modules:

- ***Load Source*** – it loads source code in the simulator. It also finds syntax errors in the source;
- ***Cycles per Instruction*** – it defines the number of the cycles needed for the execution of the current instruction; it also determines the type, the number of the operands, and the address mode.
- ***Coupling of Instructions*** - it defines whether two instructions can be coupled and executed simultaneously avoiding data hazard.
- ***Branch Prediction*** - this program simulates the functioning of branch prediction. If there is a jump, a branch, or a call instruction, branch prediction proceeds to predict if the branch shall be taken or not.

The simulator of IA-32 architecture consists of four base modules. One of them is ***Coupling of instructions***. It determines whether two instructions can be coupled and executed together avoiding data hazards. In this paper the algorithm of module Coupling of instructions is represented.

Each program instruction gives itself the following information:

- a source operand;
- a destination operand.
- is it a simple instruction or not;
- is it an ALU instruction or not;

According to the decoupling rules, only hardware-executed instructions can be decoupled and executed together. This kind of instructions are: MOV, LEA, PUSH, POP; ALU type instructions like ADD, ADC, SUB, SBB, AND, OR, XOR, NOT.

ALU type instructions need two sources. Intel Pentium has mixed architecture - between RISC and CISC. IA-32 instructions use two data operands, so instructions like ADD use the same operand for destination and first source. That is why it is necessary to know if it is an ALU instruction or not.

The algorithm for detection of data dependency gives information about dependency of a two-neighbor instruction. It is more interesting to find data dependency between more than two-neighbor instructions. The second algorithm detects this kind of data dependency.

This information is described as follows:

```
struct  instruction_info
{  char destination[4];
   char source[4];
   unsigned char s_type:1;
   unsigned  char  alu_type:1;
 };
```

If the instruction is hardware-executed, s_type is 1; if the instruction is one of ADD, ADC, SUB, SBB, AND, OR, XOR, NOT, alu_type is 1. *Destination* and *source* are destination/source data or register.

The algorithm works as follows:

```
struct instruction_info prev, next;
```

```
/* prev, next keep information about two neighbors instructions */
int i, n;              // n is number of instructions
unsigned char *buffer;     // the result buffer
*buffer = 0;        // first instruction is independent
buffer = new char [n];
for (i=1; i<n; i++)
 { /* determining the information for the  previous instruction i-1; prev stores this information */
   /* determining the information for next instruction i; next stores this information */
 if(strcmp(prev.destination,next.source))buffer[I]=1;
   else *(buffer+i) = 0;
 if((next.alu_type)&&strcmp(prev.destination ,next.destination) buffer[I] = 1;
}
delete [] buffer;
```

The real program verifies if the source and destination registers are identical. For example, registers EAX and AX are identical but the registers AH and AL is not.

This algorithm is realized by a function in the Coupling of instructions module, which searches for independent instructions to decouple.

## 5. CONCLUSION

Instruction-level parallelism makes it possible to execute more than one instruction per cycle. Today's processors use more than one pipeline, which means that they have superscalar architecture.

Instruction-level parallelism increases the performance but an ideal sequence of uniform instructions is rare. The execution of one instruction often depends on the result of the previous instruction's execution. This situation is called data hazard. Data hazards reduce the architecture performance.

This paper treats the problem of detection of data hazards in superscalar execution. The algorithms of independent instruction detection are represented.

The firs algorithm is implemented in a software simulator, which represents the way the Intel Pentium Processor works. It can be used in out-of-order execution logic.

Simulating and showing all processes related with Pentium processor working at real-time is too hard task and it is not necessary for the objective of this research. The objective is creation of a simulator, which is able to show the pipelining in a superscalar architecture using a real existing architecture as an example.

To be created a software simulator of Intel Pentium processors' functionality, it is necessary to simulate data flow analysis. In this paper the algorithm of independent instruction detection is represented. The first version of the algorithm is used in a program module ***Coupling of instruction***, which is a part of software simulator of IA-32 architecture. The advanced algorithm is able to detect data dependency between more than two instructions. It can be used in out-of-order execution logic.

## REFERENCES:


**Hinton G., D. Sager, M. Upton, D. Boggs, D.Carmean , A. Kyker** and **P. Roussel** (2001). The Micro architecture of the Pentium 4 Processor, Intel Technology Journal Q1.

**Hlavicka J.**, (1999). Computer Architecture, CVUT Publishing house,.

**Keshava J.** and **Vl. Pentkovski** (1999). Pentium III Processor Implementation Tradeoffs, Intel Corp., Intel Technology Journal Q2.

**Zaharieva-Stoyanova, E.** (2002), Simulation Models Of Pipelining in Intel Pentium Processors, IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics, Cluj-Napoca, Romania, pp. 373-378.

**Zaharieva-Stoyanova, E.** (2002). Simulation of Pipelined Data Processing in Intel Pentium Processor, CompSysTech, Sofia.

*** (2000). A Detailed Look Inside the Intel NetBurst Micro-Architecture of the Intel Pentium 4 Processor, Intel Corporation.

*** (2001). IA-32 Intel Architecture Software Developer's Manual, Intel Corporation.

*** (1998). Pentium, NiSoft Ltd.