Improving ILP with Semantic Analyzer for Loop Unrolling in X86 Architectures

Jih-Ching Chiu, Zh-Lung Chen, and Jean Jyh-Jiun Shann

Department of Computer Science and Information Engineering
National Chiao Tung University
E-mail: jjsham@csie.nctu.edu.tw
Tel: +886-3-5731832 Fax: +886-3-5724176

ABSTRACT

In this paper, we propose an approach, called semantic analyzer for loop unrolling, which can increase ILP of loops by parsing the semantics of instructions for collecting the required information of loop unrolling. The mechanism has the functions to construct and maintain data flow graphs dynamically, and stores these graphs inside the processor. When loops occur, we can use this mechanism to solve the repeated fetching and decoding of instructions, and even to overcome the bottleneck of data dependence checking. The simulation results are used to decide the parameters of this mechanism and compare its issue rate to that of other microprocessors. The performance of our mechanism is better than that of other microprocessors. Under performance/cost consideration, our mechanism can issue 2.07 x86 instructions per cycle.

Index Terms – semantic analyzer, data flow processor, data-driven, ILP, superscalar processor, x86 architecture

1. Introduction

Researchers have discussed the use of ILP (instruction-level parallelism) to accelerate performance for more than 20 years [1]. In order to achieve high ILP, many kinds of microprocessors have been proposed, such as VLIW, superscalar and pipeline processors. However, the complex x86 instruction set makes it hard to improve the performance of x86 processors. For examples, checking instruction boundary limits the exploitation of the ILP of superscalar processors and increases hardware cost, and complex instruction semantics make it necessary to translate x86 instructions to RISC-like instructions and thus increase the burden of data dependence checking.

Most of the current CPUs are multiple issue processors. Multiple issue processors come in two flavors: superscalar, such as K7 [2], Pentium II [3]/ Pentium III [4], PowerPC [5], Alpha 21264 [6], etc., and VILW, such as Intel IA-64 and Transmeta Crusoe [7]. While a higher issue rate is often used to acquire higher processor performance, it also complicates the control and data dependencies on the processor performance at the same time. Designers may opt to eliminate dependencies during instruction issue by using register renaming and speculative branch processing. And with out-of-order execution techniques, like scoreboarding and Tomasulo's algorithm, the results can be forwarded to the current instructions when the precedent instructions produce the results at the execution stage. By these ways, we may avoid the latency caused by writing, then reading the operands from the register file or reorder buffer to progress out-of-order execution. This can be known as the pioneers of data-driven computation. Although out-of-order execution is only implemented in function units, the performance it enhances is good. This application is very suitable in the computer architectures of traditional von Neumann machines.

By applying the data-driven conception to the decoder, we can get the features of loops dynamically by constructing dataflow graph in decode stage, and parse the behavior of the loops, such as the *loop-entrance data* that are the result operands produced by the instructions prior to a loop and used by the instructions in the loop, and the *loop-exit data* which are the result operands produced by a loop and may be used by the instructions next to the loop. By the operations of the Pumping-Data Unit [8] that is designed in another potion of our project, which can get a large amount of relative data simultaneously and label the operands of instructions by separate tags to approach that the loops are unrolled.

The remaining parts of this paper are organized as follows. In Section 2, we describe

* Correspondence should be sent to the third author.

** This paper presents partial result of a research project financed by NSC of R.O.C. under contract no. NSC89-2213-E-009-066.

the data driven concept in microprocessor design, the processing of data flow computers, and its weaknesses. In Section 3, we describe our design details. In Section 4, we do the simulation to decide the parameters of our semantic analyzer and compare the performance of our design with other microprocessors. Finally, in Section 5, we give our conclusions of this work.

2. Backgrounds

It is well known that, to optimize a program for speedup, efforts should be focused on the regions where the payoff is the greatest. Loop structures in a program represent such regions.

2.1 Exploiting Loop Performance in Traditional Processors

In early pipeline processors CDC 6600 and Cray 1, loop buffers have been used to hold sequential instructions contained in a small loop. The loop buffer operates with two advantages. First, it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes if the target of a branch is within the loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer.

In the designs of Bellas and his colleagues [9], the loop cache is proposed and is placed between the CPU and I-cache, but the D-cache subsystem is not modified. Since most of the programs tend to execute frequently only a small subset of their instructions, the loop cache can be used to capture these instructions and provide them to the CPU. This scheme can be used for performance improvements provided that the hit rate in the loop cache is very high. Therefore, the loop cache has a heavy influence on the loop performance because it is specially designed for exploiting the ILP of loops. In this approach, the compiler is given the duty to select the appropriate part of the code to be placed in the loop cache. Hence the compiler acts an important role in this scheme.

Loops in pipeline and superscalar processors can use the branch prediction to cross the limitation of basic blocks. But iterations in loop must still be executed sequentially. Although VLIW processors can achieve high ILP, we need excellent compilers to recompile all original applications, and this is inconvenient for users. Lately, Intel Merced dynamically translates x86 instructions to VLIW instructions by special hardware. It results in not only more hardware complexity and long stage delay, but also low clock rate.

2.2 Exploiting Loop Performance in Data Flow Processors

The serialization of the von Neumann computing model is a serious limitation for exploiting more parallelism in superscalar microprocessors. The current CPUs make use of reservation stations to achieve similar benefits of the dataflow model. However, the execution of a program is still limited by the program counter. Dataflow computers in 1980s had their own languages and compliers, and these computers could translate a program to a dataflow graph by their compliers. Because data flow processors may label separate tags in different iterations of a loop, they can well improve ILP of loops.

The static dataflow architecture was first proposed by Dennis and Misunas in 1975 [10]. At the machine level, a dataflow graph is represented as a collection of activity templates, each containing the opcode of the represented instruction, operand slots for holding operand values, and destination address fields referring to the operand slots in subsequent activity templates that need to receive the result value. On the other hand, the typical dynamic dataflow architecture consists of a matching unit that stores and matches tokens, a program memory, an instruction-fetch unit that takes appropriate instructions from the memory, arithmetic and logic unit (ALU) to perform the instructions, and an output unit to direct the tokens to their destination [11].

The dataflow approaches is assumed that the arcs are first-in-first-out (FIFO) queues. This is accomplished by extending the basic firing rule as follows:

An enabled node is fired if there is no token on any of its output arcs and when the resources are available.

By this way, this dataflow model has a serious deficiency for handling loop. Consecutive iterations of a loop can only be pipelined, which limits the amount of parallelism that can be exploited. The performance of a dataflow machine will significantly increase when loop iterations and subprogram invocations can be preceded in parallel. To achieve this, each loop iteration or subprogram invocation should be able to be executed as a separate instance of a re-entrant subgraph. This replication, however, is only conceptual.

Consequently, the dataflow mechanisms are limited by FIFO queues because the same instructions in different iterations of the loop must be executed sequentially due to the FIFO queues. And the dataflow computers need their own language and complier, and can only work for small-sized programs. Accordingly, we must translate x86 instructions to dataflow instructions for compatibility, and it is complicated. In this paper, we will associate the conceptions of data-driven to capture the loop semantics, and the techniques of the loop cache and the loop buffer to exploit high ILP of loops in general codes.

3. Design of the Semantic Analyzer for Loop Unrolling

The traditional fetch units are limited to one branch prediction per cycle and can therefore fetch one basic block per cycle or up to the maximum instruction fetch width, whichever is smaller. As Figure 1 shows, loops occupy a large part of general programs, and hence, it is very profitable to exploit ILP of loops. Therefore, we intend to increase ILP and the issue rate of loops by parsing the semantics of instructions in order to collecting the required information for loop unrolling. We name this method as the semantic analyzer for loop unrolling.

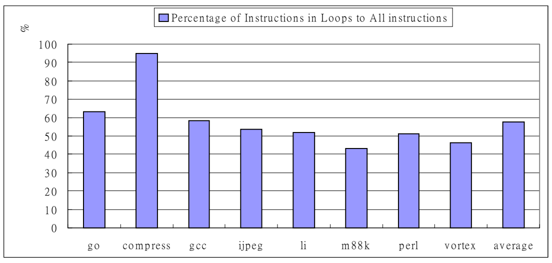

Figure 1. The percentage of the instructions in loops to all instructions.

In a program trace, if the program counter of the current instruction equals to that of a previous instruction in the instruction stream, then there may be a *loop*. For reasonably simplifying the mechanism of the semantic analyzer for loop unrolling, a loop that can be recognized by our mechanism must satisfy all of the following four conditions:

(1) The loop has only one branch instruction whose target address is not equal to any of the program counters of the instructions in this loop.

(2) The number of iterations of the loop must be explicitly known as one of the following two types of examples:

for (i = 0; i < 100; i++) … Style-1 Loop
for (i = 0; i < n; i++) … Style-2 Loop

where the variable *i* is called a *loop-counter*.

(3) We must consider the influence caused by the memory overlap of arrays in a loop because this will limit the maximum unrolling numbers of the loop. The *head-value of an array* means the starting location of the array in the memory. The *stride of an array* indicates the difference of index values in this array. The *strides of the arrays* in the loop must be all positive or all negative. And the loop must satisfy the condition that if the arrays are permuted according to the increasing (or decreasing) order of the *head-values*, then the *stride* order of the permuted arrays must also be increasing (or decreasing).

(4) There is no function call in the loop.

3.1 Structure of the Semantic Analyzer for Loop Unrolling

The computational kernel of a data-driven machine is applied to a traditional x86 microprocessor in our project. Figure 2 shows the detailed block diagram of the Data Flow Engine. The *Loop Semantic Analyzer* can parses an instruction fetched by the fetch unit, and store the parsed information, such as the operand's data dependences, the data-flow graph of these sequent instructions, and the loop structure. When the *Loop Semantic Analyzer* detects a loop, it will send the parameters of requested data to the *Pumping-Data Unit* and set up loop unrolling environment at the *Match Unit*. Once the data enters the Match Unit in parallel and is matched with its partner, we will put the corresponding instructions of the data flow graph to data prepare unit to wait for execution. Then the functional unit will send the result data back to the pumping-data unit via Register File, in order to continuously precede the operations that are dependent on the result data.

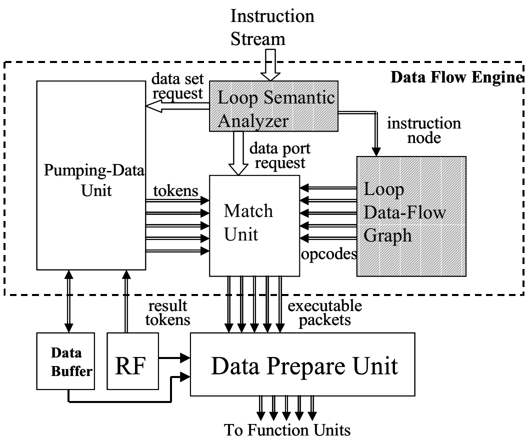

Figure 2. The detailed block diagram of the Data Flow Engine.

In the paper, The *Loop Semantic Analyzer* is our design goal. The parsing procedures of the typical loops are designed to organize it.

3.3 Loop Semantic Analyzer

In order to clearly describe how we get the semantics in the instruction stream, we will describe the necessary tables for storing the semantics of instructions, called semantic tables, and the procedures for parsing the instructions.

3.2.1 Semantic Tables

There are four types of sematic tables in the *Loop Semantic Analyzer* for storing the semantics of instructions, called Instruction Table, Source Table, Result Table, and Dependence Table.

Instruction Table

The Instruction Table, as shown in Figure 3, is used to store the format of every instruction in the instruction stream. Each entry in the Instruction Table consists of a program counter of the x86 instruction (PC), an opcode, and three other fields that are used to point to the corresponding positions of the source operands in the Source Table and the destination operand in the Result Table.

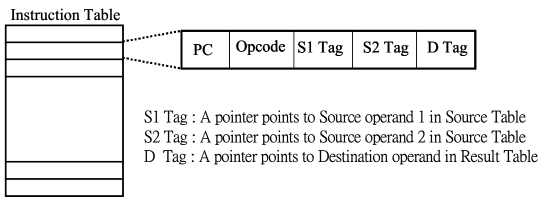
Instruction Table
PC
Opcode
S1 Tag
S2 Tag
D Tag
S1 Tag : A pointer points to Source operand 1 in Source Table
S2 Tag : A pointer points to Source operand 2 in Source Table
D Tag : A pointer points to Destination operand in Result Table

Figure 3. Structure of the entries in the Instruction Table.

Source Table

Each cycle when an instruction flows into the semantic analyzer, the source operands of the instruction will be keep in Source Table. The entry in the Source Table consists of a mark bit, the source operand, and a field that points to the position of the source operand in the Instruction Table. Figure 4 shows the structure of the entries in the Source Table. The *mark bit* is used to mark if the source operand is dependent with the destination operand of a previous instruction.

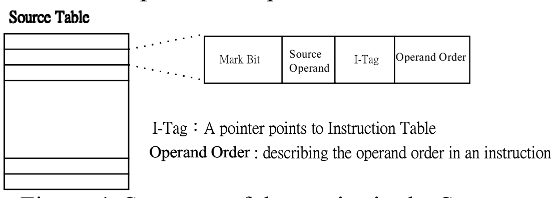
Source Table
Mark Bit
Source Operand
I-Tag
Operand Order
I-Tag : A pointer points to Instruction Table
Operand Order : describing the operand order in an instruction

Figure 4. Structure of the entries in the Source Table.

Result Table

When the source operands of an instruction are kept in Source Table, the destination operand of the instruction is also kept in the Result Table. As Figure 5 shows, each entry in the Result Table consists of a tag of the instruction in Instruction Table that owns the destination operand, and the destination operand.

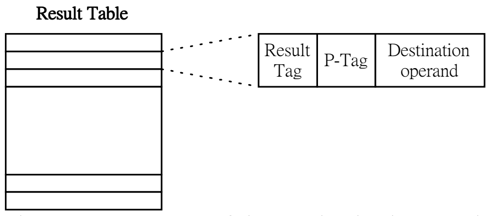

Figure 5. Structure of the entries in the Result Table.

Dependence Table

When the source operand of an instruction is dependent with the destination operand of a previous instruction, the program counters of the instruction and the previous instruction are kept in the Dependence Table. Each entry in the Dependence Table, as Figure 6 shows, consists of a program counter of a previous instruction that produces the dependent data (PC-P), a program counter of the instruction that consumes the dependent data (PC-C), and a field that points to the position of the source operand in the Parsing Table that needs the dependent data.

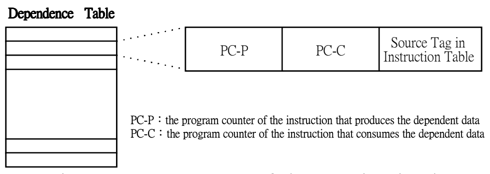
Dependence Table
PC-P
PC-C
Source Tag in Instruction Table
PC-P : the program counter of the instruction that produces the dependent data
PC-C : the program counter of the instruction that consumes the dependent data

Figure 6. Structure of the entries in the Dependence Table.

3.2.2 Parsing Procedures of Instructions

When every instruction flows into the semantic analyzer for loop unrolling, it must be processed by the following procedure：

PROCEDURE 1

==================================

1st：From the decoder, we can get the relational information of the current instruction, such as source operands, destination operand, types of the source and destination operands, and opcode, and put the instruction into the Instruction Table. Then go to step 2.

2nd：We put the source operands of the current instruction into Source Table. If the operand has appeared in Result

Table, we will set the mark bit of the operand in Source Table and go to step 3；else go to step 4.

3rd：Because we must keep the dependent relation between the instructions in Dependence Table, we will write “the tag” of the current instruction and “the tag” of a previous instruction whose destination operand is dependent with the source operand of the current instruction into Dependence Table. Then go to step 4.

4th：We put the destination operand of the current instruction into Result Table. If the destination operand has appeared in Result Table, we will overwrite the tag of the same destination operand with the new tag of the current instruction.

When the mechanism detects a loop, it will enter the following procedure：.

PROCEDURE 2

```
=================================
1st：In Source Table and Result Table, the operands that belong to the range
     between the start-PC and the end-PC of the current loop are selected out.
2nd：The operands that are selected out and do not have mark bit in Source Table
     are regarded as entry-loop-data. Besides, the operands that are selected out
     in Result Table are regarded as exit-loop-data.
3rd：Compute the maximum unrolling number of the loop;
4th：if(inst-pc == start-pc in one of four loop frames) then
         current loop frame= the loop frame that is matched;
         command the fetcher to stop fetching instructions from cache;
         The loops are unrolled by labeling operands of instructions in current
         loop frame with different tags at first iteration;
     else    // first time to meet this loop
         find one of four loop frames to be the current loop frame with FIFO method;
         store the instructions of the loop and dependent relations into the
         current loop frame;
         store the start-pc and end-pc of the loop into current loop frame;
         the instruction is executed the same as that in superscalar processors;
     end if
5th：count the number of instructions in this loop, and store the inst-num in
     current loop frame；
6th：if( the instruction matched in inst-queue is not ‘L’ type) then
         the instruction matched in inst-queue become ‘L’ node；
     end if
```

3.3 Loop Data-Flow Graphs

In order to maintain the loop data-flow graphs of the instruction stream, we construct the loop graph storage to provide the space to store the instructions and the relational information of the loop. As Figure 7 shows, the loop graph storage consists of several *function frames*. Each *function frame* is used to store the relative information in a function, such as the program counters of the function and the caller. Each function frame also consists of one *inst-queue* and several *loop frames*. The *inst-queue* of the function frame is used to record the program counter of every instruction in the function. Besides, each loop frame consists of a *Loop Instruction Reference Table*, a *Source Table*, a *Result Table*, a *Dependence Table*, and some fields that store the relational information of the loop, such as the starting program counter, the ending program counter, execution times, etc. The *Source Table*, *Result Table*, and *Dependence Table* are used to store the dependence relations between instructions in a loop as described previously. In addition, the *Loop Instruction Reference Table* stores the program counters and types of the instructions in a loop.

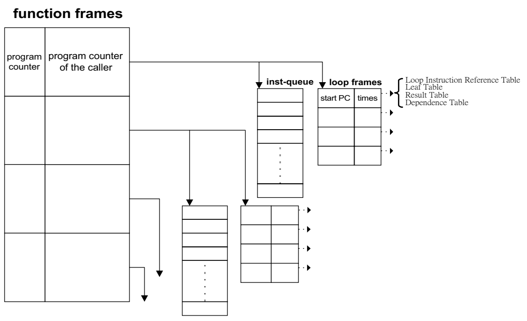

Figure 7. The structure of the loop graph storage.

When our design detects that the program counter of current instruction is equal to the starting program counter of a loop frame, we send “*Loop Instruction Reference Table*”, “*Source Table*”, “*Result Table*”, and “*Dependence Table*” to the Pumping-Data Unit for loop unrolling at the first iteration. On the other hand, when we meet a loop the first time, we will send the four tables in the loop frame to the Pumping-Data Unit for loop unrolling at the third iteration.

4. Simulations and Analyses

In this section, we describe the simulation environment and present the simulation results of our semantic analyzer.

4.1 Simulation Environment

We use trace-driven technique for our simulation. The benchmark programs are compiled using GNU C compiler. Then the compiled benchmark programs are executed on a computer running the Linux operating system. The traces of the benchmark programs are extracted using Linux *ptrace()* system call. These traces are then fed into the simulator.

The benchmark programs we use are from SPECint95, which includes *go*, *m88ksim*, *gcc*, *compress*, *li*, *jpeg*, *perl*, and *vortex*. These benchmark programs represent the characteristics of most software applications.

4.2 Simulation of the Loop Styles Handled by the Proposed Mechanism

The loop style 1 in our handling range is a loop whose *loop-counter* is a constant variable, for example, the constant 100 in "*for (i = 0; i < 100; i++)*". Besides, the loop style 2 in our handling range is a loop whose *loop-counter* is a simple variable, for example, the variable *n* in "*for (i = 0; i < n; i++)*".

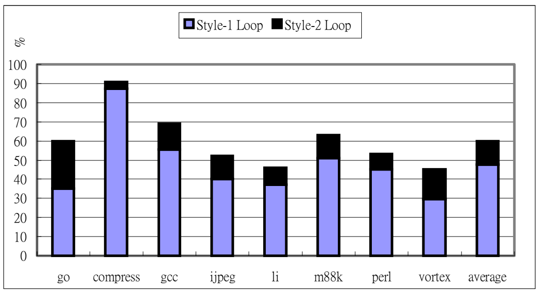

Figure 8. The percentage of style-1&2 loops to all loops.

The percentages of style-1 and style-2 loops to all loops are shown in Figure 8. The average percentage of style-1 loops to all loops is 47% and that of style-1&2 is 60%. From the preliminary simulation results, the loops that we want to deal with occupy a large part of all loops.

Furthermore, we simulate the percentage of the instructions in loops handled by our design to all instructions, as shown in Figure 9.

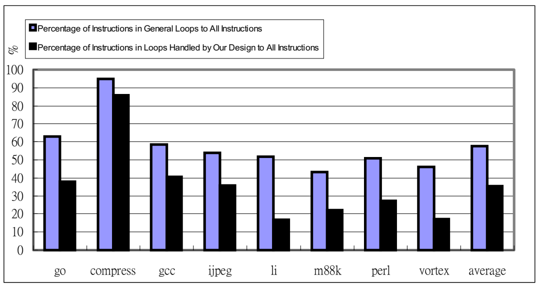

Figure 9. The percentage of the instructions in general loops or in loops handled by our design to all instructions

4.3 Simulation of the Features of Loops

In Section 3, we mentioned that the instructions in a loop are stored into the loop frame for loop unrolling. Figure 10 shows the cumulative distribution of the x86 instruction numbers in a loop. From this figure, we observe that 64-instruction loops could cover the most common (or frequent) cases of loops. Nearly 90 percents of the loops have less than or equal to 64 instructions. Besides, if the program counter of the current instruction equals to the program counter of a previous instruction in the *inst-queue*, we could know that there may be a loop. Therefore, from the simulation result we find that 64-entry *inst-queue* is a good choice under performance/cost consideration.

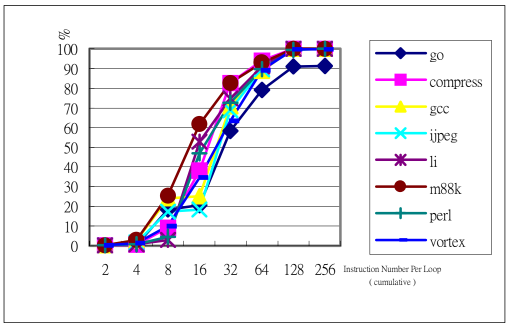

Figure 10. Cumulative distribution of instruction numbers in a loop.

Figure 11 shows the cumulative distribution of the depths of nested loops in the x86 instruction stream. The x-axis represents different depths of nested loops, ranging from 1 to 8. The number 1 at the x-axis means that the loop is not a nested loop. In other words, there are no other loops within the loop. The percentage of the loops that at most have the fixed depths to all loops is shown in the y-axis. The eight lines represent the different benchmark programs in SPECint95.

From Figure 11, we observe that most common (or frequent) loops are not the nested loops. Furthermore, the depth of nearly 97 percents of loops are less than or equals to 2. Therefore, we know that simple loops occupy a large part of general codes and 2 or 4 *loop frames* within a *function frame* is a good choice under performance/cost consideration.

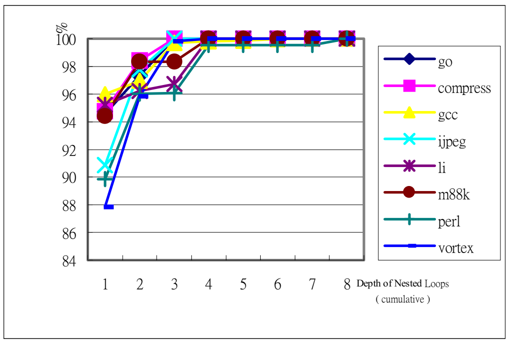

Figure 11. Cumulative distribution of the depths of nested loops.

4.4 Performance Analyses

In this subsection, we compare the x86 instruction issue rate of our mechanism to that of other microprocessors. In our simulator, the following assumptions are made：

1. The fetch unit of our mechanism fetches only one x86 instruction per cycle. But the fetch units of Pentium, Pentium MMX, and K6 series fetch two x86 instructions per cycle; that of P6 series fetches three x86 instructions per cycle; and that of K5 series fetches four x86 instructions per cycle.
2. The function units may execute any kinds of instructions in a clock cycle.
3. Every instruction can be hit in the cache.

The results are shown in Figure 12. In this figure, we apply the semantic analyzer for loop unrolling to the two simulation models, noted as *our choice* and *Ideal*. In *our choice*, we assume that the simulation model has 7 issue degree, 3 unrolling degree, 64-entry inst-queue, and 3 loop frames within a function frame. On the other hand, the *Ideal* model means unlimited issue degree, unlimited unrolling degree, unlimited-entry inst-queue, and unlimited loop frames within a function frame.

From this figure, we find that the performance of *Ideal* is excellent and its issue rate is 2.3 x86 instructions per cycle. Besides, the issue rate of *our choice* also surpasses all the other microprocessors and achieve at 2.07 x86 instructions per cycle. This simulation shows that the semantic analyzer for loop unrolling mechanism have its benefits.

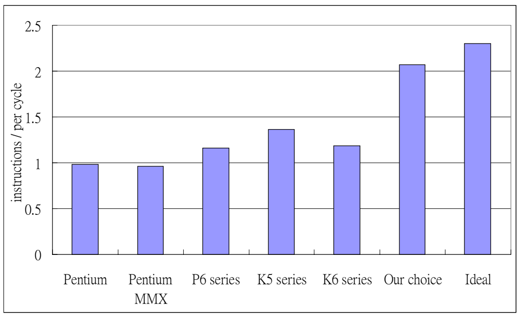

Figure12. Issue rate comparison.

5. Conclusions and Future Works

In this paper, the conception of data-driven computation is applied to design the x86 microprocessors, and we name the hardware the semantic analyzer for loop unrolling. The semantic analyzer is designed to detect loops in the instruction stream by parsing the rich semantics of instructions and stores the instructions in the loops to the loop frames for loop unrolling. The mechanism has the functions to construct and maintain data flow graphs dynamically, and stores these graphs inside the processor. When loops occur, we can use this mechanism to solve the repeated fetching and decoding of instructions, and even to overcome the bottleneck of data dependence checking.

From the evaluations, the average percentage of instructions in the loops that can be handled by the proposed mechanism is 35% to all instructions. Therefore, shows that it is worthy to exploit the ILP of loops. The simulation results show that the issue rate of the semantic analyzer for loop unrolling mechanism is higher than that of the current commercial superscalar microprocessors.

In our design, we only recognize the loops of which repeated times may be explicitly known. However, the ILP of loops could be wider by unrolling other kinds of loops, such as the structures of software pipeline and dynamic linking. The structure of *software pipeline* means a loop that has dependent semantics between iterations, such as “A[i] = A[i-1] + k”. The structure of *dynamic linking* means a loop that we can not explicitly know its iteration times, such as “while (ptr != NULL)”. Thus, researches of parsing other kinds of loops are the future works of this paper.

[References]

[1] B.R. Rau and J.A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," in J. Supercomputing, Vol. 7, No. 1/2, pp. 9-50, 1993

[2] AMD Corporation, "AMD Athlon(TM) Processor Architecture," August 23, 1999

[3] Intel Corporation, "Pentium II Processor Developer's Manual," October 1997

[4] Intel Corporation, "Pentium III Processors — Datasheets," May 2000

[5] S.P. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC microprocessor," in IEEE Micro, Volume: 14 Issue: 5, pp. 8, Oct. 1994

[6] R.E. Kessler, "The Alpha 21264 microprocessor," in IEEE Micro, Volume: 19 Issue: 2, pp. 24 –36, March-April 1999

[7] Transmeta Corporation, "The Technology Behind Crusoe(TM) processors," January 2000

[8] 國科會產學合作計畫，具資料流引擎之x86 微處理機設計 (編號 NSC89-2213-E-009-066)

[9] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Energy and Performance Improvements in Microprocessor Design using a Loop Cache," in ICCD, 1999

[10] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," in Proceedings of the 2nd Annual Symposium on Computer Architecture, pp. 126-131, Houston, TX, January 1975

[11] E.J. Lerner, "Data-flow Architecture," in IEEE Spectrum, pp. 57-62, April 1984