Toward More Advanced Usage of Instruction Level Parallelism by a Very Large Data Path Processor Architecture

Hidehiko TANAKA
Department of Electrical Engineering
The University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan

Abstract

The architectural performance gain of a micro processor is going to saturate because of the small gain of instruction level parallelism. In this paper, we discuss the design points and some tentative solutions to overcome this bottleneck and propose a processor architecture called Very Large Data Path. This architecture broadens the window of instruction analysis to extract 10 times of parallel gain compared with the conventional superscaler processors. This paper discusses the system elements and shows some preliminary evaluation results.

1 Introduction

Personal Computers are now the general business tools for almost all of the business persons. Workstations and servers are used to support the large computation and files. The computer usage will grow more and spread to the home use and to the daily-life usage.

The load of computers will grow more as multimedia handling and agent-based processing will be used heavily in such wide-spread usage environment by the people at large. So, the needs of high performance of computers will not saturate these 10years.

The major technologies for the high performance computing are the parallel processing and the high performance processor. Parallel processing is expected to be the major architecutural technology which is essential for such high performance as greater than Tera Flops. However, it needs different programming style and has poor compatibility with the conventional program code up to this time. In the case of high performance processor, the architectural performance gain of superscalar and VLIW is going to saturate, though we can still expect some performance improvement by the finer device technology.

These two technologies are orthogonal in the sense that the high performance processor can be the element of parallel processing systems.

This paper discusses the feasibility study of a still higher performance processor which is needed to be developed as the core element of coming information society while keeping the code compatibility with the conventional sequential code, and proposes a processor architecture called Very Large Data Path(VLDP) with some preliminary evaluation results.

2 Requirements and Design Problems

2.1 Requirements of the New Processor Design

We discuss the processor architecture design problems upon such conditions as follows.

1. general purpose processor
2. code-upward compatible with some conventional processor
3. target device technology of $0.10\mu m$
4. average architectural performance gain of more than 10

Special purpose instructions such as the one for multi-media processing are very powerful. However, we concentrate our discussion not to the instruction set but to the general architecture for instruction handling (condition 1). We can incorporate such instructions into our architecture preserving the upward compatibility.

The condition 2 makes our attention focus on the design of a single processor. The condition 3 permits us to use the clock speed at the range of 1000MHz, the available gates of more than 10^8 and the chip size of 20mm $\times$ 20mm. It will be around the year 2007 when such device technology is available (3 or 4 generation ahead from now: 0.35, 0.25, 0.18, 0.13, 0.10 μm).

The condition 4 shows such target as the average architectural performance gain of more than 10. This means that we need to extract the instruction-level parallelism at least in the order of 10. If we succeed to get this order of gain, the total average performance of a processor chip will be 10 GIPS.

2.2 Design Problems

When we try to design some processor architecture on the conditions listed in the previous subsection, we face such design problems as follows.

1. Branch Prediction
   A branch instruction comes out every 4 to 6 instructions of conventional object program code. As a conditional branch instruction forks the instruction stream at the point, we need to fetch both instruction streams when we want to examin

the codes succeeding the branch operation, unless we can predict correctly the result of branch operation. The preciseness of this prediciton is about 90% by the latest technology. To reduce the number of fetched code, we need a very accurate prediction system.

2. Instruction Prefetch
   In order to ensure the instruction-level parallelism of more than 10, we need to be able to fetch many instruction codes during one clock interval. That is, due to the data dependency among instruction codes, every instruction is not always executable at the same time as the previous instructions. This means we need to prefetch instructions at the fetch speed of more than several times of the number of parallel executable codes.

3. Data Prefetch
   As executable codes require some operand data to be processed, we need to supply the data before the instruction codes begin to process them. Otherwise, we should wait for the arrival of data and are faced to the degradation of performance. This means we need to prefetch the data before their utilization if the data is placed in some slow main memory. The hit ratio of 95% is not enough for the processing environment where 10s of instruction codes are executed at the same time. Accordingly, we need more efficient cache mechanism by incorporating explicit prefetch with the support of dynamic/static data flow analysis.

4. Data Path Design
   When we are provided with several parallel executable instruction codes, the next step is to allocate to them the resources such as ALU, data bus time slot, the result registers, etc. Usually the result registers are often used by the succeeding operations, supplying the contents to the succeeding operations by bypassing the registers improves the performance of exection. If the registers are used only for keeping data of the succeeding operations, they can be replaced by simpler latch flip flops. This replacement removes the necessity of registers and the tentative store operations to the main memory in some case. Accordingly, this reduces the traffice between processor and main memory. The use of latches requires the configuration of a network among ALUs.

5. Recompilation
   We want to extract enough instruction-level parallelism to fill up the executable code register through analysing the original machine code. However, usually it is no easy task to extract such levels of parallelism from machine code only. In that case, we can expect the assistance of compiler by recompiling the source and obtaining the more necessary information for the analysis of parallelism such as branch prediction and data dependency analysis.

3 Proposal of VLDP Architecture

We propose a processor architecture called Very Large Data Path. This architecture is an extension of Superscaler in terms of the instruction level parallel architecture.

The block diagram is shown in figure 1.

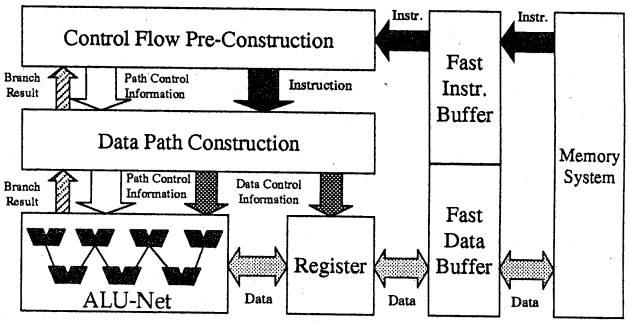

Figure 1: **Very Large Data Path Processor Architecture**

The processor is made of 4 modules: Control Flow Pre-Construction module, Data Path Construction module, ALU-Net module and Buffer module. Control Flow Pre-Construction module fetches instructions from instruction buffer as much speculatively as possible within the limit of hardware resouce, predicts branches, optimizes the code block and makes up the instructions set within which there are no control dependency.

This behaviour is shown in figure 2. This Control Flow Pre-Construction module maintains the control flow path while fetching instructions. When the fetched instruction is a branch, the speculative fetch is done based on the probability of branch prediction. As the result, a set of control flow pathes is made up as figure 2, where some of the pathes are fetched speculatively at the corresponding branch, and the others are not. Within these pathes, the unnecessary pathes are discarded when a real path is settled as the result of instruction execution.

In this figure, an arrow corresponds to an instruction. Solid lines show the fetched ones. Broken lines the ones which are not fetched at the time. Value xxx means the branch probability at the point, of which dependency is shown by thin lines.

Data Path Construction module analyses the codes to be executed, resolves their data dependency and allocates them to the ALU-Net which is composed of a collection of ALUs and a connecting network among ALUs.

Buffer module is made of 3 units: Fast Instruction Buffer, Fast Data Buffer and Registers. Depending on the status of execution, these buffers store the most likely instructions and data which will be used in near future, by using a cache like mechanism enhanced by an explicit rearrangement algorithm of data(instruction) location.

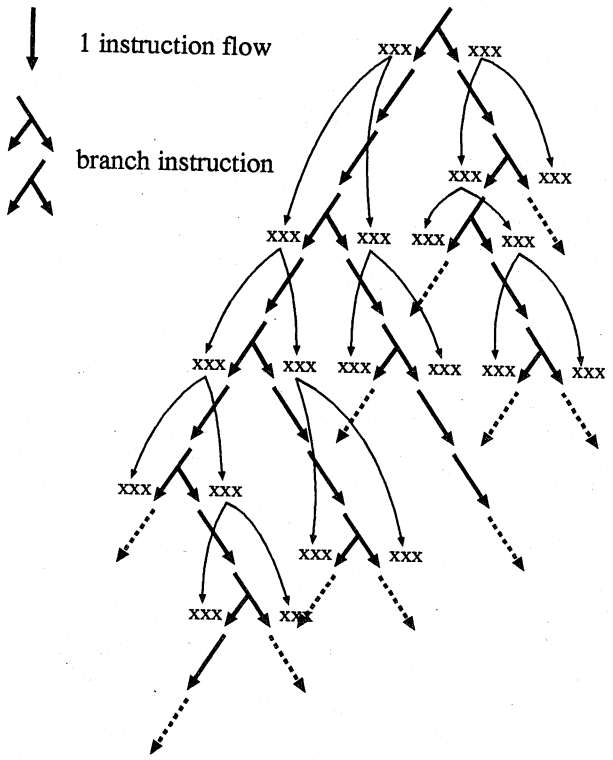

Figure 2: **Image of Control Flow Pre-Construction**

The Control Flow Pre-Construction module fetches instructions in the unit of a basic block of which average size is 4 through 7 for conventional programs.

In the VLDP architecture, we have several elemental mechanisms which determines the basic performance of this architecture. In the next section, we discuss these mechanisms and show the evaluation result of each elemental mechanism.

4 Elemental Algorithms and Their Performance

4.1 Branch Prediction

Up to this time, Branch Histry Table and Branch Target Buffer are developed ([1, 2, 3, 5]). However, the performance of prediction is not enough. We propose to add other mechanisms such as explicit branch control called XXXcc and CCC, and an expansion of Branch Target Address Cache.

XXXcc is a mechanism of starting the branch prediction at the execution point of such instruction that determines the condition code for the coming conditional branch. This mechanism is expected to improve the prediction correctness dynamically.

CCC is the condition code cache which supports the XXXcc mechanism by hardware. That is, we provide a cache memory of which data element is,

PC(XXXcc), PC(Bicc), Condition,

where PC(XXXcc) designates the program counter of the instruction which determines the value of condition code, PC(Bicc) is the program counter of the branch instruction, and Condition designates the condition of the branch instruction.

We try to broaden the distance between PC(XXXcc) and PC(Bicc) as much as possible, so that the branch condition is known as fast as possible. Using these mechanism, we can expect the performance improvement of branch prediction in the order of 1-4 percent(reference [6]). This absolute value is not large. However, this value at the 92% level of correctness is meaningfull to reduce the needed number of speculative code prefetches.

Regarding the Branch instructions which use register-deferred addressing such as Jump and Link instructions, conventional Branch Target Address Cache can exploit only 70% of predection accuracy for branch target addresses. We can improve this prediction accuracy up to 80-90% by the expansions of Branch Target Address Cache (reference [10]).

4.2 Data Prefetch

We need to supply data to each instruction fast enough for their execution. However, static data prefetch mechanism proposed so far is not enough. While the dynamic data prefetch mechanism up to this time ([4]) predicts the access address of data by assuming that the address difference of 2 successive accesses by the same instruction in a loop does not change, the prediction correctness is not enough and it is not equipped with a precise generation mechanism of the prefetch timing.

We propose an extended stride method which permits the prefetch only when the length of succeeding strides matches to the previous one. By this extention, we can improve the hit ratio in the order of 10 to 18 % as shown in figure 3 and 4.

For the predition of prefetch timing, we propose to use the time interval of succeeding 2 data accesses of the same instruction as the basic interval and to adjust the number of predicted accesses to be issued adapting to the memory latency. This mechanism is effective to reduce the volume of necessary hardware compared with LA-PC of the paper [4].

When failed to predict, the memory access is cancelled not to increase the memory access traffic.

The global performance improvement of this mechanism is several to 30 % where 100% improvement of performance is the one of such system that secondary cache and main memory access latency is assumed to be 1 cycle.

4.3 Dynamic Code Optimization

Control Flow Pre-Construction module fetches basic blocks of a program and synthesizes an execution pack of codes by optimizing the execution of n stages of basic blocks. This execution pack of codes is the unit of parallel execution by the ALU-Net module.

If we have 2 forks at the every end of a basic block, the total number of execution pathes is $2^{(n-1)}$. At the execution stage, one of the pathes is selected.

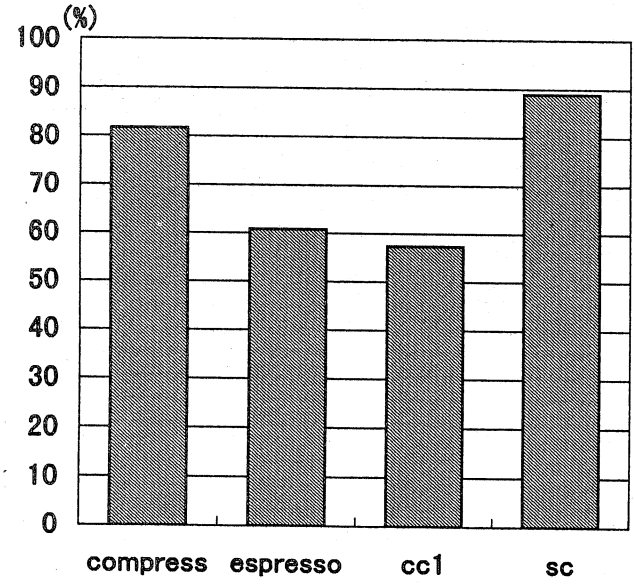

Figure 3: **Hit ratio of the stride method**

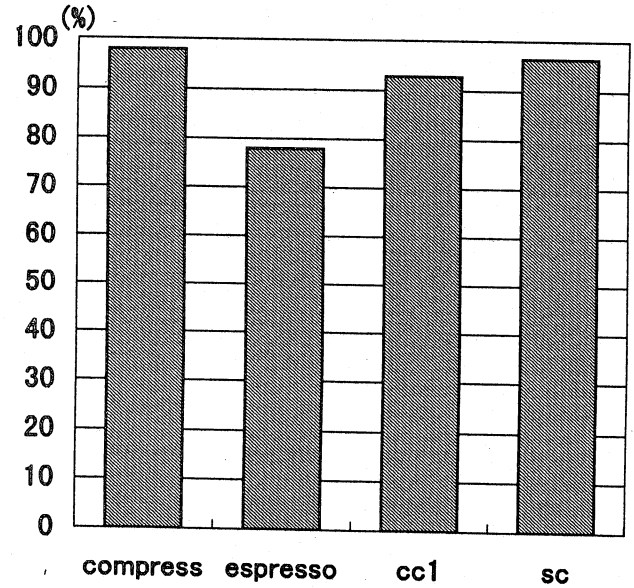

Figure 4: **Improved hit ratio of the extended stride method**

The dynamic code optimization is done for each path of this concatenation of basic blocks. When we have enough execution resources, this optimization is equivalent to make up a data flow graph.

Figure 5 shows the effect of this optimization when the number of fetch stage n is 5.

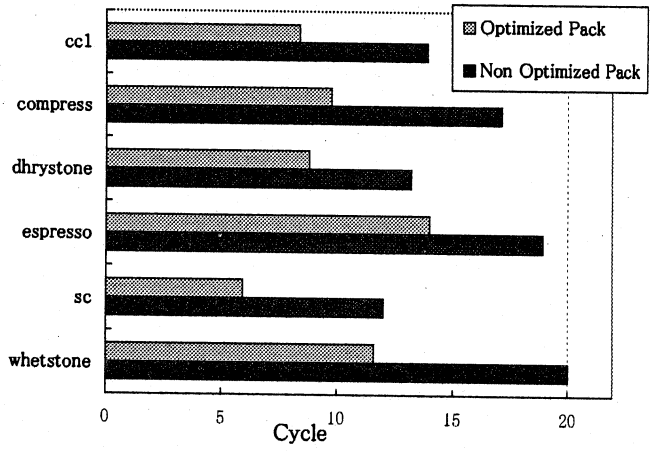

Figure 5: **Effect of Dynamic Code Optimization**

The ratio of the cycles of Non Optimized Pack to the one of Optimized Pack shows the parallelism at this level. It is from 1.5 to 2.0. This is the gain when we use the pack as the execution unit. Accordingly, the value correspondes to the gain of VLIW system, which is the case where a pack is generated and assigned to a very long instruction by the compiler. This means that the sequential execution only of basic blocks does not generate enough parallelism. We need to integrate the other kinds of paralleism such as the pipeline execution of packs, the inter-pack parallelism and the independent instruction streams which are expected to exist on the other path than the one which is focused attention.

4.4 Code Fetch Efficiency

When we fetch n stages of basic blocks, the total number of basic blocks is $2^n - 1$ at the maximum. For the usual programs, the effective number of basic blocks is less than 50% of this maximun.

Table 1 shows the percentage of effective codes with the number of fetch stages as the parameter. The application programs are the one of SPECint 92.

For example, the percentage of effective instruction codes is about 16 through 31 and the total number of fetched instructions is 100 through 150(not shown here), when the number of fetch stages is 5.

Though this percentage is a little bit still high for real implementation, we can expect to improve it fairy much by incorporating the branch prediction including register-deferred jump instructions.

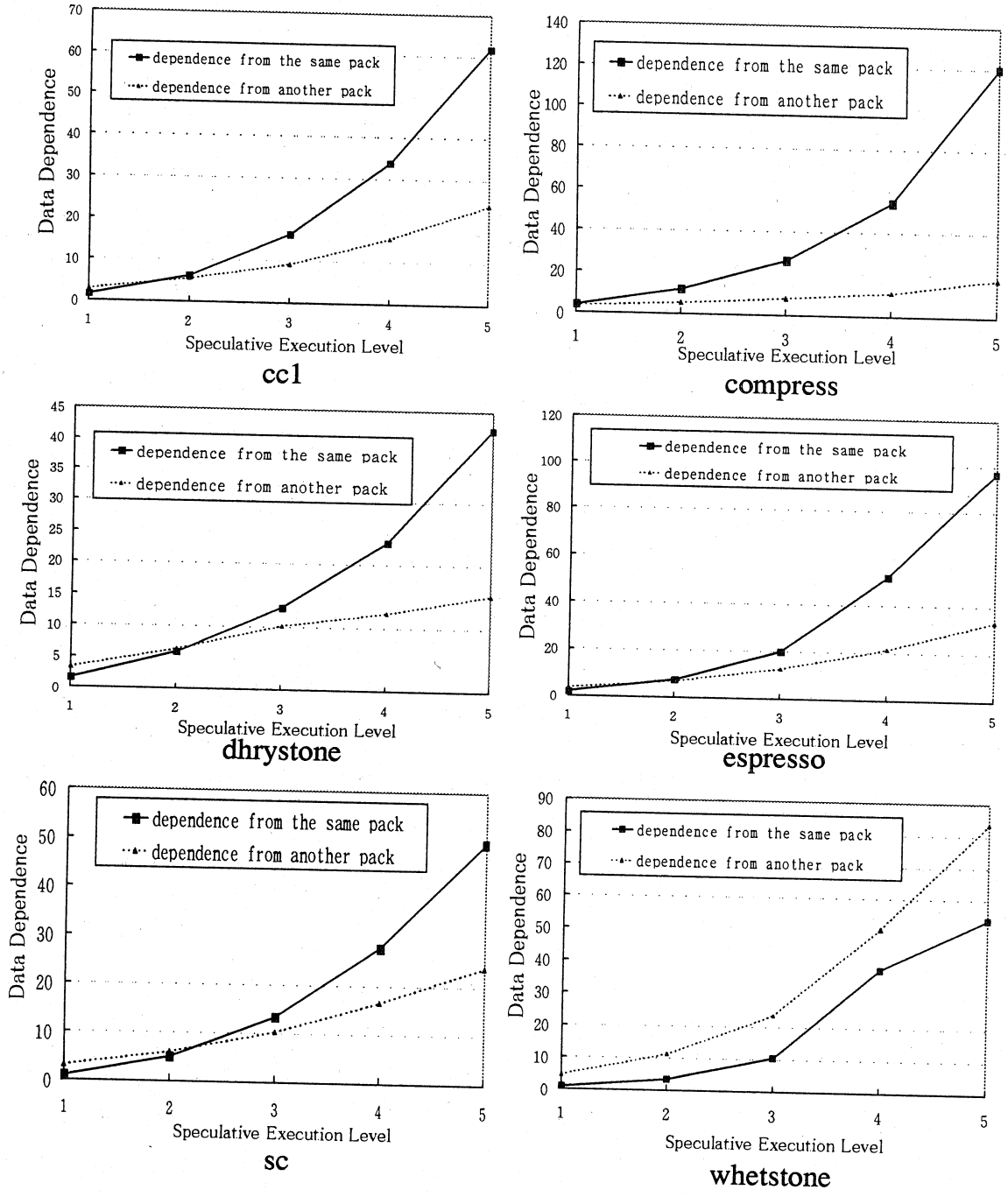

Figure 6: **Intra/Inter Pack Data Dependency**

Table 1: **Percentage of Effective Code**

program	number of fetch stages				
	1	2	3	4	5
cc1	96	68	46	30	19
compress	98	77	54	37	22
dhrystone	98	73	56	42	31
espresso	95	66	43	26	16
sc	95	64	43	30	21
whetstone	95	72	44	22	18

4.5 Memory Access Traffic by Operand Fetch

When we execute a pack of instructions, we need to feed the operand data from registers and main memory. If the number of execution codes in each pack is one, all of the operand should be provided by registers and main memory because the processed result by the previous instruction must be stored back to a register or main memory.

However, if the number of execution codes is greater than 1, we can use directly the output of some execution as the input of other instruction execution by inserting latches between ALUs. This is usefull to reduce the total traffic between registers and data path, and between main memory and data path.

Figure 6 shows the examples of the effect, where the number of executed codes is measured in terms of the fetched basic blocks in a pack, which is designated as speculative execution level in the figure.

As the result, the data traffic inside of the data path comes up to 86.3 through 61.8 % of the total data traffic when the value of execution level is 5. So, the outside traffic such as between registers and data path, and between main memory and data path is only 14 through 38 %.

This ratio of inside traffic becomes larger as the value of execution level grows. Figure 7 shows the average number of this ratio for the SPECint 92 programs.

5 Evaluation of VLDP Architecture

5.1 Instruction Level Parallelism

When we analyze program codes by using a small window size which determines the region of program code for the target of analysis, we can not get so much instruction level parallelism. For example, the average parallelism of a basic block is 1.47 to 1.92 for the SPECint 92 programs.

However, if we analyze them with a very large window size, we can expect more parallelism, which is shown in figure 8.

The vertical axis(speedup) shows the relative speed compared with the scalar execution of the programs. This is the result when we assume that we can designate the operand addresses of all Load and Store instructions before their execution and predict the conditional branch with 100% accuracy. In other word, this speed up is the upper bound of parallel gain.

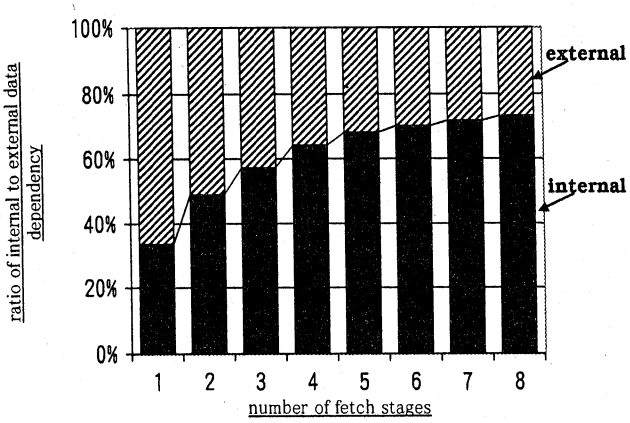

Figure 7: **Change of Inside Traffic Ratio**

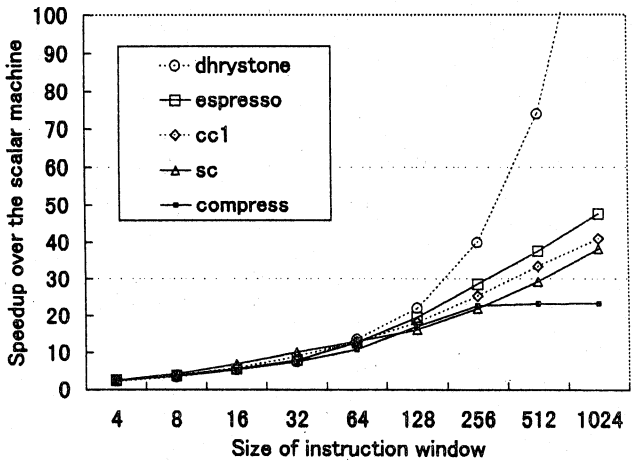

Figure 8: **Parallel Gain of Performance**

Accordingly, we can expect 10 times of parallel gain with the window size of 32 to 64 if we use all of the apriori information about the programs by profiling for example.

5.2 Upper Limit of Speed Up

Table 2 shows the upper limit of the speed up when we have enough execution resources, i.e. infinite size of instruction window.

Table 2: **Upper Limit of Speed Up**

program	upper limit of speed up
cc1	62.3
compress	31.4
dhrystone	232
espresso	125
sc	82.0
whetstone	3.11

As shown in this table, we can expect more than 30 times of speed up gain compared with the conventional scalar processors except the one of whetstone. As the program of whetstone does not have enough intrinsic parallelism, the parallel speed up gain is much limited. However, we can expect more than 15 times of performance compared with the superscalar processors of which speed up gain is 2 or so at present.

6 Conclusion

The performance of a superscaler processor is going to saturate. However, there is a possibility of much more instruction level parallelism for conventional programs, if we analyze enough number of instruction codes at run time as well as at the compile time.

This paper showed the possibility of such parallel gain by introducing a Very Large Data Path architecture which is one of the candidate to extract such parallel gain, and discussed some key mechanisms of this architecture.

Though this analysis is not enough to show the real performance and there remaines a lot of technical problems such as evaluation under the more realistic conditions for branch prediction, consideration of runtime overhead, and design of new instruction set, I think this paper could show the possibility of more performance improvement by instruction-level parallelism. I hope we will have many active researches in this area toward the new processor architecture for the coming age.

Acknowledgement

I would like to express my thanks to the member of my laboratory for their research efforts: T. Nakamura for his branch algorithm improvement, K. Kise for his evaluation of maximum performance limit and Y. Ajima for his prefetch mechanism. This research is partly supported by the Grant-In-Aid by Ministry of Education (07458052).

References

[1] J. E. Smith, "A study of Branch Prediction Strategies," *Proc. of the 8th Intl. Symp. on Computer Architecture*, Vol. 8, pp. 135-148, 1981.

[2] R. Nair, "Optimal 2-Bit Branch Predictors," *IEEE Transactions on Computers*, Vol. 44, No. 5, pp. 698-702, 1995.

[3] T. Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *Intl. Symp. Computer Architecture*, Vol. 20, pp. 257-266, 1993.

[4] T. F. Chen, J. L. Bear: "A Performance Study of Software and Hardware Data Prefetching," *Proc. of 21st International Symposium on Computer Architecture*, pp. 223-232, 1994.

[5] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proc. of the 19th Intl. Symp. on Computer Architecture*, Vol.19, pp.46-57, 1992.

[6] T. Nakamura, H. Tsuji and H. Tanaka, "Feasibility Study of Explicit Branch Control and Proposal of the Control Mechanism," *53rd Annual Conference of Information Processing Society of Japan*, No. 4F-2, pp. 115-116 (Sept. 1996)(in Japanese).

[7] P. P. Chang and N. J. Warter, S. A. Mahlk, W. Y. Chen and W. W. Hwu, "Three Architectural Models for Compiler-Controlled Speculative Execution," *IEEE Transactions on Computers*, Vol. 44, No. 4, pp. 481-494, 1995.

[8] H. Tanaka, "Let's Reconsider the Computer Architecture Now," *Information Processing Society of Japan, Workshop of Computer Architecture*, ARCH 108-6, No. 91, pp. 30-40,1994(in Japanese).

[9] T. Nakamura, K. Kise, H. Tsuji, Y. Ajima, and H. Tanaka, "Feasibility Study of A Very Large Data Path Processor Architecture," *Information Processing Society of Japan, Workshop of Computer Architecture*, ARCH 124-3, No. 61, pp. 13-18, 1997(in Japanese).

[10] T. Nakamura, K. Kise, H. Tsuji, Y. Ajima, and H. Tanaka, "A Study of Branch Target Address Prediction," *55th Annual Conference of Information Processing Society of Japan*, No. 3F-5(September 1997) (in Japanese).