Pseudo Vector Processor based on Register-Windowed Superscalar Pipeline

NAKAZAWA, Kisaburo
NAKAMURA, Hiroshi
IMORI, Hiromitsu
KAWABE, Shun*

Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1, Tennodai, Tsukuba, Ibaraki, 305, Japan
* Kanagawa Works, HITACHI Ltd.
1, Horiyamashita, Hadano, Kanagawa, 259-13, Japan

Abstract

In this paper, we present a new architecture for *high-speed pseudo vector processor* based on a superscalar pipeline. Without using cache memory, the proposed architecture is able to overcome penalty of memory access latency by introducing *register windows* with *register preloading* and *pipelined memory*. One outstanding feature of the proposed architecture is that it is *upward compatible* with existing scalar architectures. Performance evaluation of the proposed architecture using the Livermore Loop Kernels shows over 6 times higher performance than a usual superscalar processor and 1.2 times higher performance than a hypothetical extended model with cache prefetching technique with a memory access latency of 20 CPU clock cycles. List vectors are also effectively handled in a similar architecture.

1. Introduction

Motivation

Over the past several years, technological and architectural advances have realized dramatic improvements in microprocessor performance with peak performances becoming comparable to mini-supercomputers. To realize high end supercomputing, much attention has been paid recently to massively parallel systems which consist of over thousands such advanced scalar processors [Zor92]. However, performance of such massively parallel systems cannot be very high unless each node processor achieves high performance. This is difficult because the speed of the attached memory system has not increased as fast as processor speed [HJ91]. The growing gap between processor and memory speed is a serious problem.

Caches [Smi82a] are widely utilized in order to reduce the penalty of memory access latency and close the growing gap. Advanced processors can realize their potential high performance only when their caches work effectively [CP90] [Jou90]. However, especially in engineering/ scientific applications with large amount of data, data caches are not effective because such applications may have some spatial locality but little temporal locality and caches cannot keep all the data to be used. Although tremendously large data are usually accessed in fairly regular patterns, caches contribute little except for the prefetch of continuously allocated data. This problem is reported, for example, in [SW90]. The same problem arises in massively parallel supercomputers [Moy91].

Enhanced caches such as prefetched caches or lock-up free caches [Kro81] [SF91] have been proposed to reduce memory access penalties. In cache prefetching, cache blocks are prefetched before they are needed by hardware [BC91] [Smi82a] [Jou90] or software [KL91]. However, there still exist the serious problems of extra main memory traffic and increased cache traffic. Fetching unnecessary data causes extra memory traffic. Furthermore, in the worst case, cache traffic can become twice as much as memory traffic. This is due to the prefetch of extra data from lower level memory to cache, the read/write for internal processor execution, and the write back to lower memory. Therefore, expensive multi-ported caches have to be provided to match the required throughput, as described in [KL91].

Strategy

We propose a new architecture, *pseudo vector processor*, which reduces the penalty of memory access latency and realizes high-speed vector processing on advanced superscalar processors.

The problem of extra cache traffic is inevitable if caches are used only as prefetch data buffers. Vector processors avoid this problem by bringing data into vector registers directly without going through data caches, and by implementing the effective technique of chaining.

Comparisons between vector processors and superscalar processors indicate that there are some similarities in their execution of arithmetic operations. Both provide multiple pipelined functional units which can be scheduled to be fully utilized. Several vector instructions are executed simultaneously by chaining in vector processors, while several scalar instructions are also pipelined and executed simultaneously in multi-functional and arithmetic pipelined superscalar processors. Therefore, superscalar processors have ability to do simple vector processing if each scalar instruction is regarded as a *vertical micro-instruction for vector processing*. The technique of concurrent execution and data forwarding in superscalar processors can realize naturally chaining which is one of the most effective mechanism in vector processors.

However, the essential difference between two processors lies in the ability to supply data. In vector processors, data transfers between main memory and vector registers are realized *in a pipelined fashion*. Memory accesses are overlapped with processor arithmetic operations, which

exhibit *preload features*. Since sufficient *vector registers* and multiple load/store pipelines are provided, memory throughput is extremely high and the penalty of memory access latency is well hidden in vector processors. On the contrary, data transfers from/to main memory are not pipelined in most of the recent superscalar processors. As their cache systems are not effective in scientific /engineering applications, memory throughput is low while data access latency is high. The heavy penalty of data accesses seriously degrades the performance of superscalar processors.

Therefore, if the following three mechanisms are implemented, superscalar processors will be able to do economical vector processing.

- large number of floating-point registers
- enhanced data preload feature
- pipelined memory

Our strategy is to satisfy these requirements without serious changes in the instruction set architectures. It is difficult to increase the number of registers without changing the instruction formats because the register fields in instructions are limited. We have overcome this difficulty by providing sufficient physical floating point registers and splitting them into several sets, or *register-windows*. This extension requires only a few additional instructions. In order to implement *register preloading*, a few instructions are introduced and software-based preloading becomes available. Though *pseudo pipelined memory* with multiple interleaved memory banks is expensive, this is necessary for high speed vector processing. As a result, the proposed architecture holds *upward compatibility* with non-enhanced processors. This distinguishes it from other works. Although the idea of register windows is not new [Sit79] [Lam82] and has been already implemented in RISC-I [PS81] and the SPARC processor [Sun89], their register windows were introduced for the purpose of reducing the overhead of procedure calls and returns. Our purpose is entirely different, and we provide different mechanisms from theirs.

2. Principle of Pseudo Vector Processing

Register Extension with Upward Compatibility

The proposed architecture can be implemented as an extension of existing scalar architectures. The key ideas are to provide sufficient physical floating point registers which are divided into several *windows* and to introduce new instructions for *window-change*, *preload*, and *poststore*. Among the windows, just one window is allowed to be active for usual instruction execution and this active window (current window) is changed by the window-change instruction. Each window has a full set of logical floating point registers identified by the register specifiers in the instructions. Physical floating point registers are identified by the active window pointer in PSW and the register specifier in the instructions. Therefore, ordinary instructions need not specify which window to use. While ordinary arithmetic instructions can only use registers in the active window, *preload instruction*s load data into registers of the window next to the active one (hidden window) and *poststore instruction*s store data from registers of the previous window. Since the length of the register field in the instruction format can specify registers in just one window, the number of registers in a single window is the same as that in the original scalar architecture. Therefore, multiple sets of windows can be provided without changing the instruction formats. In this way, more scalar registers are introduced while keeping upward compatibility with existing scalar architectures.

Phase Pipelining on Register Window

The most time consuming parts in engineering/scientific applications are usually in the form of loop iterations.

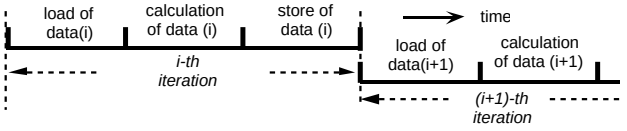
By providing pipelined memory, memory requests can be issued continuously without waiting for the return of the requested data. Each iteration loop is *conceptually* divided into three phases: data load, calculation, and result store. As shown in Figure 1-(a), these phases are usually executed sequentially, that is, the load phase for (i+1)-th iteration begins after the store phase of i-th iteration has finished. By using the well-known *software pipelining* technique, instructions from different loop iterations are interleaved and the three phases of different iterations are executed in parallel as shown in Figure 1-(b). However, in executions like that of Figure 1-(b), the register space is too limited by the register fields in the instructions and is common to all the phases executed in parallel, which leads to the interference between phases. A single phase cannot utilize full set of registers independently from other phases, and therefore, the efficiency of *loop unrolling* is suppressed.

In the proposed *phase pipelining* technique, preload, calculation, and poststore phases are executed concurrently. The registers used in each phase belong to different register spaces as shown in Figure 1-(c). This is different point from the usual software pipelining. Therefore, we can make the best use of loop unrolling within a single window.

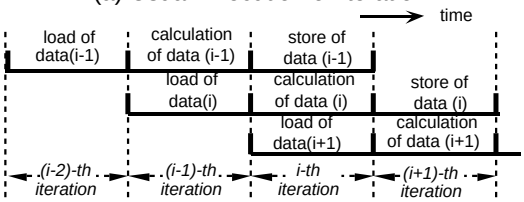
In Figure 1-(c), suppose that the active window is 'window j' in the i-th iteration. Three phases are executed in parallel as follows. In the preload phase, data for the next iteration 'data(i+1)' are preloaded into the registers in the next window 'window j+1'. In the calculation phase, arithmetic operations use the registers in the current active window 'window j'. In the poststore phase, the calculated data in the previous iteration 'data(i-1)' are stored from the registers in the previous window 'window j-1'. From the view point of register space, the data required for i-th iteration are preloaded into registers of 'window j' when the active window is j-1. Next, the active window is changed into 'window j' and the preloaded data in 'window j' are used for the calculation. After that, the active window is changed into 'window j+1' and the calculated results in 'window j' are stored by poststore instructions from 'window j'.

Real performance, of course, depends on the detail implementation of the processor and the nature of the target program. For example, if memory access latency is too large, preloaded data in the i-th iteration may not be available when the calculation of the (i+1)-th iteration requires them. In such a case, calculation must wait for the arrival of the requested data.

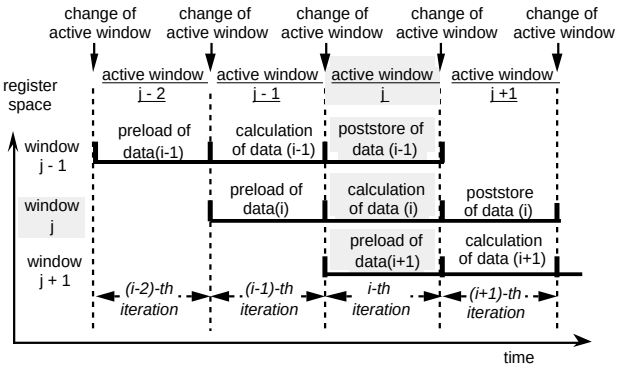
In order to avoid this, it is necessary to make the calculation phase longer. As seen from Figure 1-(c), a longer calculation phase (for example, calculation of data(i)) means a longer allowable memory latency in the next preload phase (preload of data(i+1)). Thus, longer calculation phases can alleviate the situation. Loop unrolling can be utilized to make the calculation phase be longer.



(a) Usual Execution of Iteration



(b) Usual Software Pipelining



(c) Phase Pipelining

Figure 1. Concept of Phase Pipelining

3. Architectural Extension for Register-Windowed Pseudo Vector Processor

In this section, by using an example, we describe how we extend an existing scalar architectures for *register-windowed pseudo vector processor*. The new architecture is upward compatible with the basic architecture.

3.1. Overview of Basic Scalar Architecture

The Hewlett-Packard PA-RISC 1.1 Architecture [Hew90] is selected as an example of the basic scalar architecture. We emphasize that similar extensions are possible for other commercially available RISC architectures and superscalar implementations. First, the PA-RISC 1.1 Architecture is overviewed. The extension of the architecture is explained in the next section.

- 32 thirty-two bit general registers and 32 sixty-four bit floating registers are provided. Among the floating registers, registers #0-#3 contain the floating point program status register and the exception registers. The remaining twenty-eight registers (Register #4-#31) are floating-point data registers.
- All instructions are one word (32-bits) in length. A 6-bit major opcode and several bits for extending the opcode are provided. Source and target registers are specified in 5-bit register field. Therefore, no instructions can identify over thirty-two registers.
- Memory Reference Instructions: 32-bit data is transferred between general registers and memory only by means of integer load/store instructions. Also 32 or 64-bit data are transferred between floating point registers and memory only by floating load/store instructions.
- Floating Point Operation Instructions: Besides the conventional add, subtract, multiply, and divide instructions, multiple-operation instructions are provided. For example, the operation of "FMPYADD rm1,rm2,tm,ra,ta" is "FPR[tm] <- FPR[m1] * FPR[m2]; FPR[ta] <- FPR[ta] + FPR[ra]" under the constraint that ra ≠ tm and ta ≠ rm1, rm2, tm.
- All branch instructions are delayed branches.

3.2. Architecture for Register-Windowed Pseudo Vector Processing

Floating-Point Registers and Active Window

First, the structure of floating-point registers is extended. 88 sixty-four-bit floating-point registers are provided *physically* and divided into four windows as shown in Figure 2. Each window consists of 32 registers and these registers are specified *logically* in the same manner as in the original PA-RISC architecture. As seen from Figure 2, registers are categorized into *global registers*, *overlap registers* and *local registers*. This structure is similar to SPARC's register windows [Sun89]. The floating point program status and exception registers are located in four global registers. The other four global registers are open to general usage. Four registers overlap each window. For example, register #28 in a window (say, window #1) is the same as register #8 in the next window (window #2) and is physical register #48. The overlap registers are able to deliver data of a window to the next window, which results in an efficient support for *software pipelining*. The window structure illustrated here is just an example. It is possible to extend the structure of floating-point registers in ways different from that of Figure 2.

The number of windows, global registers, or overlap registers may be selected suitably. In the following sections, performance of the proposed architecture is evaluated with this extension and remarkable effectiveness is achieved.

The current active window is pointed to by a newly introduced CFRWP (Current Floating Register Window Pointer). Since the number of windows is four in this case, two bits for the CFRWP is assigned in the program status word(PSW). The physical register number is obtained from the logical register number specified by the instruction and the value of CFRWP when the instruction is issued.

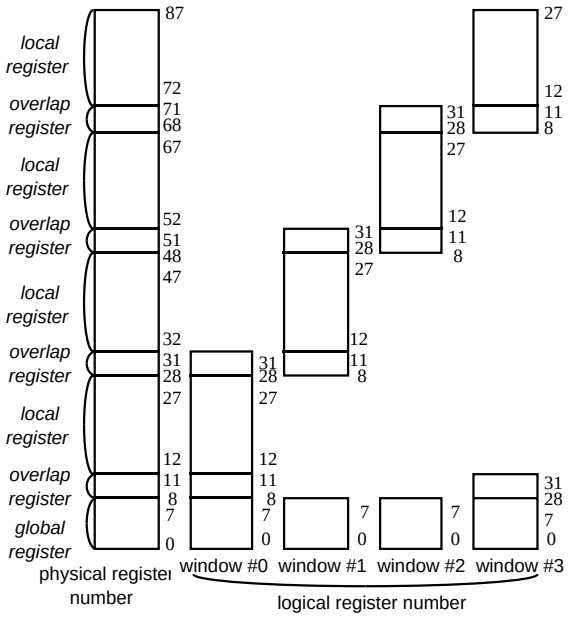


Figure 2. Structure of Floating Point Register Window

Additional Instructions

Only the following four new instructions need to be introduced. The first two instructions manage the register window while the other two instructions of preload and poststore support data transfer between main memory and floating-point registers. We need not change any instructions in the basic architecture.

- CFRWPenable: This is a privileged instruction which enable or disable the register window feature. If register window is disabled, only the registers in the window #0 are available, and the proposed new architecture becomes fully compatible with the non-windowed architecture. When the register window is enabled by this instruction, CFRWP is initially set to 0 and the current active window is 'window #0'. One bit field in PSW is assigned to indicate the status of enable/disable.
- CFRWPinc: This is a non-privileged instruction and changes the active window. CFRWP is incremented in modulo of 'the number of windows' by this instruction. No window overflow / underflow interruption is required.
- FRPreload: This instruction loads a data from memory into the specified register of the *next window*. For example, when CFRWP is set to 2, 'FRPreload r10, (m)' transfers the data into register #10 of the window #3 whose physical register number is #70. On a cache hit, the data are loaded from cache. Unlike the usual load instructions, however, on a cache miss this instruction load the data from main memory into the floating point register directly without replacing any block of data cache.
- FRPoststore: This instruction stores a data into memory from the specified register of the *previous window*. This instruction also does not replace any block of data cache on a cache miss.

4. Example of Pseudo Vector Processing

Compiled Code

Figure 3-(a) is an essential part of Lawrence Livermore Loops #1 and Figure 3-(b) is a compiled object code onto the extended architecture. For clarity of explanation, loop unrolling is not utilized and instructions for index modification and count decrement are omitted. In the PA-RISC 1.1 architecture, load/store instructions can modify index registers. Some startup codes before the loop and finishing up codes after the loop are also ignored in Figure 3-(b).

As seen from Figure 3-(b), Z(K+12) and Y(K+1) are preloaded into the next window while X(K-1) is poststored from the previous window. Calculation of X(K-1) which is a part of the previous iteration is moved into this iteration by software pipelining. Figure 3-(c) illustrates the timing of instruction execution where parallel issues and parallel executions of the memory accesses and floating-point arithmetic instructions are supported by a *superscalar scheme*. In Figure 3-(c), instructions using window j are underlined. This indicates that the FRPreload in the (i-1)-th

```
      Do 1 K = 1, n
1     X(K) = Q + Y(K) * (R * Z(K+10) + T * Z(K+11))
```

(a) Lawrence Livermore Loop #1

```
Loop:  FMPYADD r8, r5, r13,   r4, r10
            : r13 <- Z(K+10) * R
            : r10 <-  X(K-1) = ((Z(K+9) * R + Z(K+10) * T ) * Y(K)) +
Q
       FRPreload r28 Z(K+12)  : r28 <- Z(K+12)
       FMULT r28, r6, r14     : r14 <- Z(K+11) * T
       FRPreload r27 Y(K+1)   : r27 <- Y(K+1)
       FADD r13,  r14, r13    : r13 <- Z(K+10) * R + Z(K+11) * T
       FRPoststore r30 X(K-1) : store X(K-1) from r30
       FMULT r13, r27, r30    : r30<-(Z(K+10)*R+Z(K+11)*T) *Y(K)
       CFRWPinc               : increment CFRWP
       Branch                 : Branch to Loop
```

(b) Object Code on Extended Architecture

```
     multiply              add              preload / poststore
-------------------------------------------------------------------
          % (i-1)-th iteration, active window is (j-1)
1:   r13 <- Z(i+9) * R     r10 <- X(i-2)          FRPreload Z(i+11)
2:   r14 <- Z(i+10) * T                           FRPreload Y(i)
3:                  r13<-Z(i+9)*R+Z(i+10)*T       FRPoststore X(i-2)
4:   r30<-(Z(i+9)*R+Z(i+10)*T)*Y(i)       {CFRWPinc} and {Branch}
-------------------------------------------------------------------
          % i-th iteration, active window is j
5:   r13 <- Z(i+10) * R    r10 <- X(i-1)          FRPreload Z(i+12)
6:   r14 <- Z(i+11) * T                           FRPreload Y(i+1)
7:                  r13<-Z(i+10)*R+Z(i+11)*T      FRPoststore X(i-1)
8:   r30<-(Z(i+10)*R+Z(i+11)*T)*Y(i)     {CFRWPinc} and {Branch}
-------------------------------------------------------------------
          % (i+1)-th iteration, active window is (j+1)
9:   r13 <- Z(i+11) * R    r10 <- X(i)            FRPreload Z(i+13)
10: r14 <- Z(i+12) * T                            FRPreload Y(i+2)
11:                 r13<-Z(i+11)*R+Z(i+12)*T      FRPoststore X(i)
12: r30<-(Z(i+11)*R+Z(i+12)*T)*Y(i)     {CFRWPinc} and {Branch}
-------------------------------------------------------------------
```

(c) Timing of Compiled Code Execution

Figure 3. Compiled Code of Lawrence Livermore Loop #1 on Register Window

iteration, arithmetic instructions in the i-th iteration, and the FRPoststore in the (i+1)-th iteration always use the registers in window j.

Permitted Memory Access Latency

We now consider the minimum allowable latency time from the request of the preload instruction to the use of the requested data. This time interval is called the *permitted memory access latency* or the *permitted latency* in short. If the actual memory access latency is shorter than this permitted latency, no extra waiting cycles occur and processor performance is not degraded by memory access penalty. As seen from Figure 3-(c), the permitted latency is 5 in this code because Z(i+11) is requested in the first cycle and used in the sixth cycle. If loop unrolling is utilized, the cycle time for single iteration becomes longer and the permitted latency also becomes longer. Of course, the longer is the permitted latency, the less severe is the requirement for memory access penalty. The other way to alleviate the severe requirement of the permitted latency is to introduce of an *extended preload instruction* which loads a data into the next to the next window. We will also evaluate this extension.

Register Allocation

There are some data which are used in more than two iterations. For example, Z(i+11) is used both in the i-th and the (i+1)-th iteration. The same is true for X(i) because the calculation of X(i) is moved into the next iteration. The proposed architecture can handle such data effectively by allocating them in the overlap registers. Data used in all the iterations such as Q, R, and T are also handled effectively if they are allocated on the global registers. Figure 4 shows the register allocation of the register window of Figure 3.

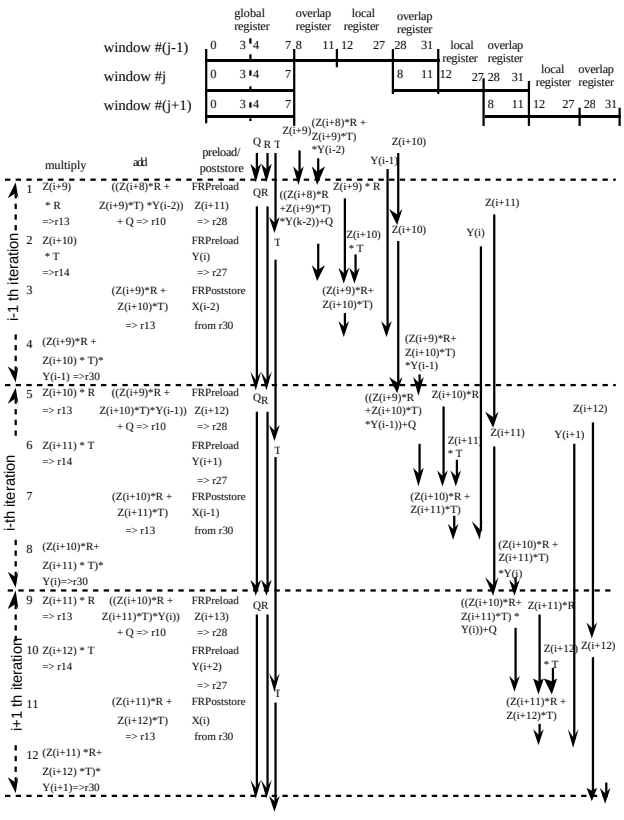


Figure 4. Register Allocation of Lawrence Livermore Loop #1 on Register Window

5. Evaluation Framework

5.1. Evaluated Model

In addition to the proposed hardware architecture model, we have evaluated the following types of processor models for comparison. All of these models have a similar superscalar pipeline scheme.

<Original> Original PA-RISC 1.1 Architecture with an Ordinary Cache

<Cache-Prefetch> PA-RISC 1.1 Architecture with Pipelined Memory and Prefetch to Cache Instruction

<Proposed>Proposed Extended PA-RISC 1.1 Architecture

<Proposed-E> Proposed Extended PA-RISC 1.1 Architecture with Extended Preload Instructions

<Ideal> Hypothetical PA-RISC 1.1 Architecture without any Cache Miss

Here, <Proposed> and <Proposed-E> are the proposed architectures for pseudo vector processing. Extended preload instruction is available in <Proposed-E>. <Original> is the original PA-RISC 1.1 model with a conventional demand-fetch data cache. In this model executions are stalled when cache misses occur. <Cache-Prefetch> is chosen as a typical model with cache prefetch feature. In this model, a prefetch to cache instruction is introduced and memory is pipelined. Prefetch instructions are performed in pipeline and cause no stalls in processor operations. The cache is assumed to be fully associative. <Ideal> is a hypothetical processor which is the ideal case of <Original> such that no cache miss occurs in the processor. The characteristics of these processors are summarized in Table 1.

5.2. Benchmarks

We have tested these models on several vector computations. In this paper, the evaluation results using the Livermore Loop Kernels #1 ~ #14 of 64 bit data are reported. We have increased the number of iterations in order to make the problem size closer to real engineering/scientific applications. Therefore, the data size becomes much larger than data cache size and conventional caches contribute little except for block transfers of multiple data in a block.

5.3. Assumptions for Evaluation

We have made the following assumptions. These assumptions are common to all the processor models unless otherwise specified.

- Parallel Instruction Issue:

Two instructions are issued in every clock cycle. All instructions are divided into three categories and the two issued instructions must be selected from different categories. The first category consists of load and store

Model	Original	Cache-Prefetch	Proposed	Proposed-E	Ideal
Architecture	PA-RISC 1.1	PA-RISC 1.1 with cache prefetch	extended PA-RISC 1.1	extended PA-RISC 1.1	PA-RISC 1.1
Register	no change	no change	register window	register window	no change
Memory	not available	pipelined	pipelined	pipelined	no access
Preload / Prefetch Feature	not available	prefetch to cache instruction	preload and poststore	preload, poststore, and extended preload	not available
Cache	conventional	multi-ported and fully associative	conventional	conventional	always cache hit

Table 1. Summary of Evaluated Processor Model

instructions. Preload and poststore in <Proposed> and <Proposed-E>, and prefetch in <Cache-Prefetch> also belong to this category. The second category consists of floating point arithmetic operations, and the last category consists of branch and integer ALU operations. CFRWPinc of <Proposed> and <Proposed-E> belongs to the last category.

- In Order Instruction Issue:

All instructions are issued in order except simultaneous issues of allowable two instructions. If an instruction is stalled, all the following instructions are interlocked.

- Data Cache:

The block size of the data cache is 16B and continuous two double-precision floating-point data reside in the same block. No line conflict (collision) miss is assumed to occur. This assumption is equivalent to a fully associative cache. The capacity of cache is assumed to be insufficient to keep all the data. In other words, cold start of data cache is assumed. In <Cache-Prefetch>, data cache is multi-ported.

- Main Memory:

In <Proposed>, <Proposed-E>, and <Cache-Prefetch>, main memory access is pipelined by a single load/store pipeline and one preload/poststore instruction or prefetch to cache instruction can be issued every clock cycle. This feature is implemented, for example, on a pseudo pipelined memory with multi-interleaved banks. It is assumed that all the required data are allocated optimally in main memory so as not to cause memory bank conflicts. In <Original>, memory access is not pipelined. The memory access latency is altered during the evaluation. This latency includes the transfer time between processor and storage control unit, bank control logic, and error code corrections.

- Penalty of Data Dependency:

Since in-order issue is guaranteed, only RAW (read after write) dependency need to be considered. If an instruction (instrB) tries to read a source operand which is a result of a preceding instruction (instrA), instrB should be issued several machine cycles later than instrA. If instrA is a floating-point operation, instrB should be issued 5 cycles or more later than instrA. If instrA is a usual load instruction, instrB should be issued 2 cycles or more later than instrA on a cache hit and 'memory access latency' cycles or more later on a cache miss. If instrA is a prefetch instruction in <Cache-Prefetch> or a preload instruction in <Proposed> or <Proposed-E>, the penalty is the same as a load. In <Proposed> or <Proposed-E>, some hardware mechanisms have to be provided to synchronize arithmetic instructions with preload instructions because they operate independently and concurrently. However, this can be done by an easy extension of already available hardware logic for resolving usual data dependencies.

- Penalty of Control Dependency:

Branch instructions are delayed branches in this architecture. If a delay slot is filled by an effective instruction, then there is no penalty caused by control dependency.

- Instruction Cache:

All the required instructions are brought into the instruction cache in advance. Warm start is assumed in the instruction cache. This assumption is appropriate for the benchmark because it is a collection of simple loops.

5.4. Evaluation Methodology

We have optimized the codes for the Livermore Loop Kernels by hand and estimated the performance by simulating the execution in instruction pipeline. Estimated performances are given in FLOPS (floating point operations per second). In the optimization, the codes are unrolled as many times as possible with the available registers.

Optimized codes for different processor models differ from one another. Only the codes for <Ideal> and <Original> are the same. The code for <Cache-Prefetch> is obtained by inserting prefetch instructions into the code for <Ideal>. Since cache memory has more space than registers, prefetch to cache instructions are moved ahead enough so as not to cause data waiting delays. In the codes for <Proposed> and <Proposed-E>, although the codes are scheduled so as not to cause any data dependency, data waiting delays can still occur if memory access latency is larger than the *permitted latency*, because preload to register instructions and extended preload instructions cannot be moved ahead without limitation as is the case for prefetch instructions.

6. Evaluation Results

Figure 5 shows the performance (MFLOPS) of each processor model on the individual Livermore Loops under the conditions that the clock-rate is 100MHz, 2 instruction issue per cycle, and a memory access latency is 20 cycle (200nsec).

We have also evaluated the performance of each processor model with different the memory access latencies.

Figure 6 shows the relative performance of each model compared with <Ideal>. Here, the performance reported is the harmonic mean of Livermore Loop Kernels. Since no cache misses occur in <Ideal>, its performance is not influenced by memory access latency at all.

Figure 6-(a) shows the performance for Livermore Loop #1 ~ #12. These loops are selected because they can be vectorized in recent vector supercomputers. When memory access latency is equal to 0, the relative performance of <Cache-Prefetch> is 0.84. This degradation arises because prefetch instructions are inserted and the total cycles required is increased. The relative performance of <Proposed> and <Proposed-E> is 0.99. In the proposed structure of register window, some registers are shared by multiple windows. Therefore, in these processors, loops are occasionally not unrolled as many times as in the other models. This is the reason of this slight performance degradation.

Performance of <Original> decreases seriously when memory access latency is increased. When memory access latency is 20 CPU cycles (200nsec), which is a practical value if DRAM is used, the relative performance of <Original> falls to 0.15. <Cache-Prefetch> is 5.6 times faster than <Original>. This speed-up is due to the pipelined memory, the multi-ported cache and the sufficiently high memory/cache throughput. <Proposed> and <Proposed-E> are 6.3 times and 6.6 times faster than <Original> respectively. Furthermore, they are 1.1 times and 1.2 times faster than <Cache-Prefetch> respectively.

The next interesting observation is the performance degradation when memory access latency is increased further. As seen from Figure 6-(a), performance of <Original> is very low. In Livermore Loop #1~#12, the array element references are not dependent on the run time data. Therefore, we assume that the compiler can move prefetch instructions far ahead enough without cache pollution. This is the reason why the performance of <Cache-Prefetch> does not seem to be affected by increasing access latency. However, this assumption is not practical. Moving prefetch instructions too early is accompanied with the danger of cache pollution. Fully associative caches are also assumed in this evaluation. If caches are not fully associative, however, early prefetch may cause line conflicts. In <Proposed> and <Proposed-E>, performance degrades if memory access latency is larger than the permitted latency. However, the relative performance of <Proposed-E> remains at 0.99 when memory access latency reached 30. <Proposed-E> is superior to the others until memory access latency reaches 50.

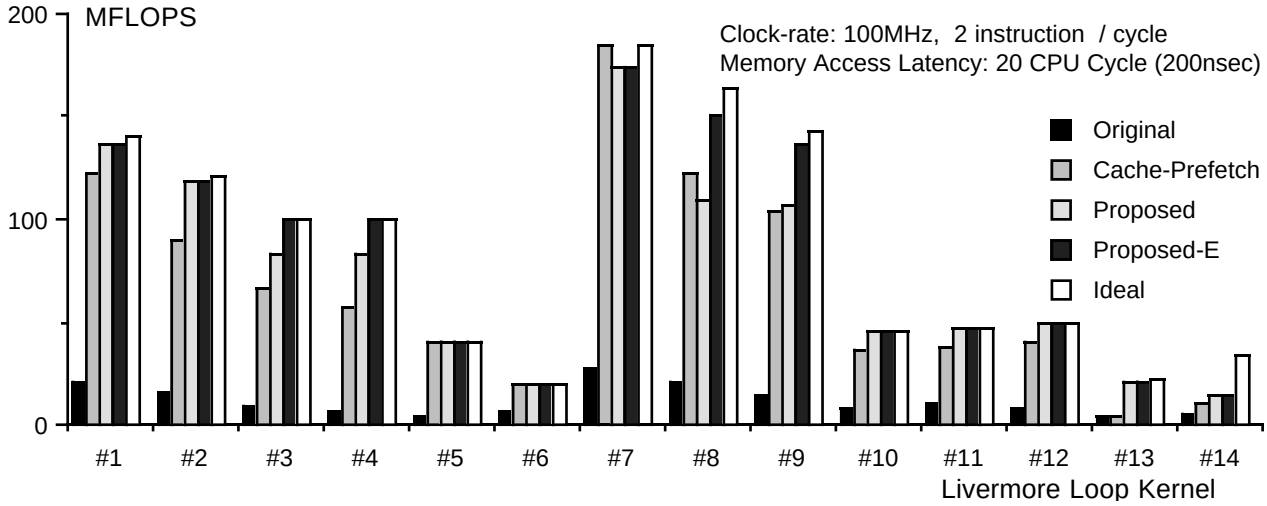
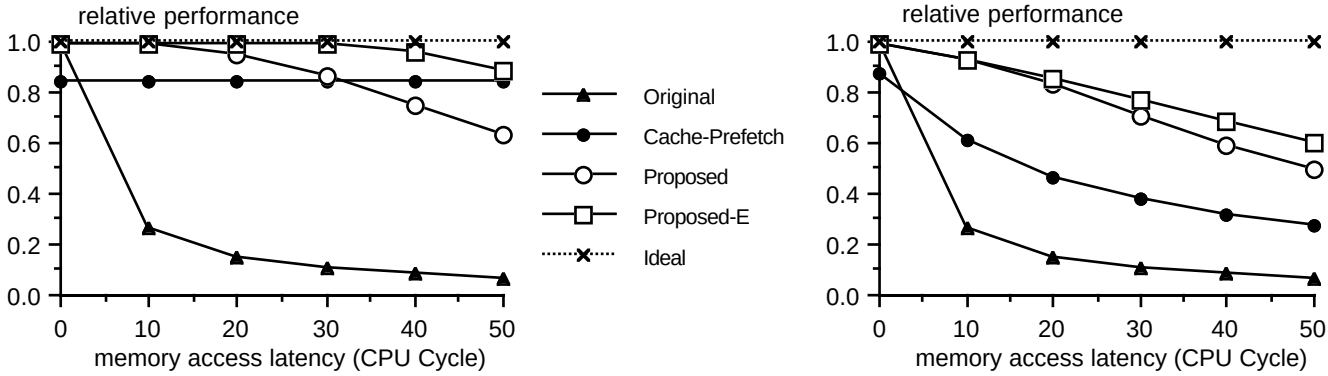


Figure 5. Performance on Livermore Loop Kernel



(a) Performance for Livermore #1 ~#12

(b) Performance for Livermore #1 ~#14

Figure 6. Relative Performance under Variable Memory Access Latency

Figure 6-(b) shows the relative performance for Livermore Loop #1~#14. In this figure, the performance reported is also the harmonic mean. Compared with Figure 6-(a), the performance of <Cache-Prefetch> degrades as memory access latency increases because the array element references in Livermore #13 and #14 are dependent on the run time data and consequently the compiler cannot move prefetch instructions ahead enough to cover the memory access latency. Due to the same reason, the *permitted latencies* in <Proposed> and <Proposed-E> are rather short in Livermore #13 and #14, which leads to the performance degradations. However, as seen from Figure 6-(b), the degradation rate of <Proposed> and <Proposed-E> is less than that of <Cache-Prefetch>. When the latency is 20 CPU cycle, <Proposed> and <Proposed-E> are 1.7 times and 1.8 times faster than <Cache-Prefetch>. In the case of 50 CPU cycle latency, they are 1.8 times and 2.2 times faster than <Cache-Prefetch> respectively. These results indicate that the proposed processor can cover the penalty of memory access very effectively.

7. Discussions

7.1. Effectiveness of Proposed Architecture

As seen from the evaluation results, the proposed architecture extended with preload, register window, and pipelined memory realize extremely effective vector processing on superscalar processors. Because of this, we called the proposed architecture as *pseudo vector processor*.

The proposed architecture may seem equivalent to an architecture with pipelined memory and non-windowed eighty-eight floating-registers. However, without sufficiently long register specifier fields in the instructions, such a large number of registers cannot be used. Our architecture have overcome this difficulty by adopting the register windows. The important point is that the physical register space is enlarged in keeping *upward compatibility* with existing scalar architectures. The proposed architecture can make the best possible use of proven scheduling techniques such as loop unrolling and software pipelining by using non active (hidden window) registers.

Register Preloading vs. Cache Prefetching

In Figure 6, the performance of <Cache-Prefetch> degrades by the insertion of extra prefetch instructions. However, this problem may be avoided by hardware-based cache prefetch technique [Smi82a] [Jou90] [BC91].

One block look-ahead policy was described in [Smi82a]. In that policy, upon referencing block i, the only potential prefetch is of block i+1. The use of *stream buffers* which provide automatic sequential prefetching was proposed in [Jou90]. These schemes are limited to continuous data accesses or where there is good program locality. If data access patterns differ from that assumed, extra cache / memory traffic is generated. In [BC91], a hardware-based prefetch with prediction was proposed. In this technique, not only continuous data references but also constant stride references are handled well. However, random accesses are not handled effectively.

Ideally, hardware-based prefetch technique may prefetch all the required data into a data cache without extra prefetch instructions. However, cache prefetching techniques still have the following disadvantages.

- Fully random accesses are not handled effectively.
- Extra cache traffic is generated as mentioned in Section 1. Expensive multi-ported data cache is required.
- Unnecessary data may be fetched into the block in the cache. Extra cache / memory traffic is generated.
- Early prefetch may cause cache pollution.
- Collision (line conflict) misses may increase unless data cache is fully associative.

These problems do not arise in the proposed register preloading architecture because the requested data are transferred directly into the specified registers.

Proposed Processor vs. Vector Processors

The performance of the proposed architecture is lower than that of vector processors because of the lesser number of load/store and arithmetic pipelines. However, the proposed architecture has the following advantages when compared with vector processors.

- In the proposed architecture, the technique of *strip-mining* is not required because vector registers are not utilized.
- When *virtual memory* is supported in vector processors, *dynamic address translation* and the handling of page-fault are difficult because multiple data are transferred by several vector load/store instructions concurrently. In the proposed architecture, virtual memory is supported as easily as in ordinary scalar processors.
- Some advantages of *unified scalar/vector floating-point operation* proposed in [JBW89] are available.
- *List vectors* are processed effectively. Traditional vector processors cannot handle list vectors efficiently without expensive hardware resources. However, the proposed architecture has some possibility for list vector handling. This will be discussed in section 7.5.
- The overhead for start-up is relatively large in vector processors. Since set-up of vector control registers is not needed on the proposed architecture, the overhead for start-up is less and $N_{\frac{1}{2}}$(the half-performance length) [HJ88] is shorter than usual vector processors. In the proposed architecture, the start-up overhead includes only preloads for the first execution and initializations of index/base registers.

7.2. Hardware Implementation

Additional hardware for register windows includes tens of floating-point registers, several bits of the PSW, translation logic to generate a physical register number from CFRWP and a logical register number in an instruction, and some extensions of the dependency managing logic. These are small in space and easily implemented on CPU chip. The clock rate is not affected. Compared with traditional vector registers, much fewer registers (only one tenth to one hundredth) are required to play the same role in the proposed architecture. Moreover, the required number of input/output ports of the registers is almost the same as in ordinary superscalar processors.

The pipelined memory is implemented using memory banks and interleaving technique as in ordinary vector processors. Although implementing a pipelined memory is expensive, a pipelined memory is still cost-effective because sufficient memory throughput is essential in vector processing where caches are not effective.

7.3. Compiler Related Issue

We have obtained the performance results by hand-compiled codes. To develop an effective compiler is our next goal.

One direction of efficient compilation is the extension of the *modulo scheduling* on rotating register files proposed in [RLTS92]. First, optimized code is generated by software pipelining under the assumption that all the physical registers are available. Then, the generated code is scheduled so as to match the register window scheme.

Another direction of compilation is suggested from our experiences of hand compilation, where codes for the proposed architecture is obtained as follows. First optimized code under windowed register is generated without preload instructions. The algorithm of register allocation is the most important in this step. Next, load instructions are changed to preload instructions and moved ahead into the previous iteration. Handling of index registers which address the data to be preloaded is the most important point in this step.

7.4. Related Works

The idea of register windows is not new [Sit79] [Lam82] and has been already introduced in RISC-I [PS81] and the SPARC processor [Sun89]. Our purpose and implementation is entirely different from these. We have introduced the register windows for the purpose of increasing the number of registers and reducing the penalty of memory access. The register window does not play the role of a stack in our architecture. Therefore, in our architecture, window overflow/underflow interrupt is not required when window circulation occurs.

A unified approach to vector and scalar computation was proposed in [JBW89] and adopted as the floating-point architecture in MultiTitan [JDBN88]. Their purpose is similar to ours. However, we did not introduce vector instructions or vector registers, and therefore, our architecture can be upward compatible with existing load/store scalar architectures.

Decoupled architecture [Smi82b] was also proposed for the purpose of reducing the penalty of off-chip memory access. Its performance evaluation is described in [SWP86]. The difference between the decoupled architecture and our register windows architecture is that the former transfers data into a queue whereas the latter transfers the requested data directly into the specified register of the hidden window. Thus, the decoupled architecture requires additional load/move instructions for data transfers between the queue and registers.

A pipelined floating-point load instruction (pfld) was introduced in i860 processor [Int89]. The load pipeline has three stages. A pfld returns the data from the address referred by the third previous pfld and the pipelined pfld instruction does not place the data in the data cache on a cache miss. This instruction also hides memory access latency. However, for a data load, the destination register and the source location in the memory are defined in different instructions. Therefore the number of the stages in load pipeline strictly affects the object codes. That is, if the number of the load pipeline stages is changed, the object codes must also be changed in order to obtain the correct computation results. Compared with i860 architecture, our architecture includes the usual waiting mechanism for requested data and successfully closes the growing gap between processor and memory speed without serious changes in the architecture.

7.5. Further Extension to Register Windows

Structure of Register Window

The number of register windows is fixed in the evaluations in this report. The number of global, overlap, and local registers are also fixed. However, the optimal structure of these registers must depend on the nature of the program to be executed. Therefore, the following extension is also worth discussing. Namely, fixed the number of physical registers but allow the compiler to arbitrarily form the logical structure of registers within the physical register space. The number of global, overlap, and local registers is changeable under the constraint that the number of total registers in one window is fixed. To implement this extension, each compiled code must reflect what logical register structure is assumed in the compilation.

Effective Handling of List Vector

The access of the *list vector* A(B(i)) is basically an indirect addressing of vectors. If the value of B(i) is available before the access of A(B(i)), there is no problem. Therefore, a mechanism of the advanced fetch of B(i) is necessary. This is enabled by introducing *register windows into general registers*.

Figure 7 illustrates the principle of list vector handling. Suppose the CFRWP is 'j' in the i-th iteration and the execution in the i-th iteration is "A(B(i)) = A(B(i)) + const.". This kind of execution are effectively handled in the following way. When CFRWP is 'j', the calculated A(B(i-1)) is poststored from the previous floating-point register window 'j-1'. Ordinary floating operations use the active floating-point register window 'j'. A(B(i+1)) is preloaded into the next floating-register register window 'j+1', and the pointer (indirect address) of B(i+2) is preloaded into the next to the next *general register window* 'j+2'. As a result, B(i) was preloaded into the general register window 'j' in the (i-2)-th iteration at first. Next, A(B(i)) was preloaded into the floating-point register window 'j' in the (i-1)-th iteration. A(B(i)) is calculated in the i-th iteration. Finally the calculated A(B(i)) is poststored from the floating-point register window 'j' in the (i+1)-th iteration. The key point in this scheme is that the preload of A(B(i)) and the poststore of A(B(i)) can utilize the value of B(i) as indirect

address which was already preloaded into the general register window 'j'.

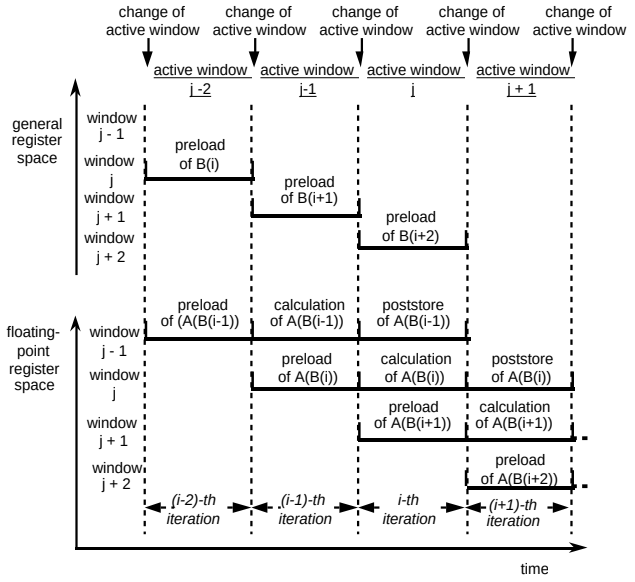


Figure 7. Principle of List Vector Handling

8. Conclusions

We have presented and discussed a new architecture for high-speed *pseudo vector processing* with a superscalar processor. The proposed architecture is able to minimize the penalty of memory access by introducing *register window* with *register preloading* and *pipelined memory*. The proposed architecture holds *upward compatibility* with existing scalar architectures. This is one of the outstanding points of this work. The performance evaluation shows that this architecture hides the penalty of memory access well. The performance of the proposed architecture is over 6 times higher than the original PA-RISC 1.1 Architecture and 1.2 times higher than the hypothetical extended model with cache prefetching technique when the penalty of memory access is 20 CPU clock cycles. We also described extensions to effectively manage list vectors.

Acknowledgements

We appreciate the valuable comments of Prof. E.Goto at Kanagawa Univ., Prof. Y.Oyanagi at Univ. of Tokyo, and Prof. I.Nakata and Prof. T.Boku at Univ. of Tsukuba. We would like to thank all the members of the GNOH group and the CP-PACS group for the many fruitful discussions. Finally, many thanks go to W.F.Wong for his helpful comments and careful revision of the manuscript.

References

[BC91] J.L.Baer and T.F.Chen, "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty", Proc. Supercomputing '91, pp176-186, 1991

[CP90] D.Callahan and A. Porterfield, "Data Cache Performance of Supercomputer Applications", Proc. Supercomputing '90, pp564-572, 1990

[HJ91] J.L.Hennessy and N.P.Jouppi, "Computer Technology and Architecture: An Evolving Interaction", IEEE Computer, Vol.24, No,9, pp18-29, 1991

[Hew90] Hewlett-Packard Company, "PA-RISC 1.1 Architecture and Instruction Set Reference Manual", Manual Part Number 09740-90039, 1990

[HJ88] R.W.Hockney, C.R.Jesshope, "Parallel Computers 2", Adam Hilger, 1988

[Int89] Intel Corp., "i860 64-Bit Microprocessor Programmer's Reference Manual", ISBN 1-55523-080-6, 1989

[Jou90] N.P.Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", Proc. 17th Int'l Symp. on Computer Architecture, pp364-373, 1990

[JBW89] N.P.Jouppi, J.Bertoni, and D.W.Wall, "A Unified Vector/Scalar Floating-Point Architecture", Proc. 3rd Int'l Conf. on Architectural Support for Programming Languages ad Operating Systems (ASPLOS-III), pp134-143, 1989

[JDBN88] N.P.Jouppi, J.Dion, D.Boggs, and M.J.K.Nielsen, "MultiTitan: Four Architecture Papers", Tech. Rept. 87/8, Digital Equipment Corporation Western Research Lab, 1988

[KL91] A.C.Klaiber, H.M.Levy, "An Architecture for Software-Controlled Data Prefetching", Proc. 18th Int'l Symp. on Computer Architecture, pp.43-53, 1991

[Kro81] D.Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization", Proc. 8th Int'l Symp. on Computer Architecture, pp81-87, 1981

[Lam82] B.W.Lampson, "Fast Procedure Calls", Proc. 1st Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-I), pp66-75, 1982

[Moy91] S.M.Moyer, "Performance of the iPSC/860 Node Architecture", Tech. Rept. IPC-TR-91-007, University of Virginia, 1991

[PS81] D.A.Patterson and C.H.Sequin, "RISC I: A Reduced Instruction Set VLSI Computer", Proc. 8th Int'l Symp. on Computer Architecture, pp.443-457, 1981

[RLTS92] B.R.Rau, M.Lee, P.P.Tirumalai, and M.S.Schlansker, "Register Allocation for Software Pipelined Loops", Proc. ACM SIGPLAN '92 Conf. on Programming Language Design and Implemenation, pp283-299, 1992

[Sit79] R.L.Sites, "How to use 1000 registers", Caltech Conf. on VLSI, 1979

[Smi82a] A.J.Smith, "Cache Memories", ACM Computing Surveys, Vol.14, No.3, pp473-530, 1982

[Smi82b] J.E.Smith, "Decoupled Access/Execute Computer Architecture", Proc. 9th Int'l Symp. on Computer Architecture, pp.112-119, 1982

[SW90] M.L.Simmons and H.J.Wasserman, "Performance Evaluation of the IBM RISC System/6000: Comparison of an Optimized Scalar Processor with Two Vector Processors", Proc. Supercomputing '90, pp132-141, 1990

[SWP86] J.E.Smith, S.Weiss, and N.Y.Pang, "A Simulation Study of Decoupled Architecture Computers", IEEE Trans. on Computers, Vol.C-35, No,8, pp.692-702, 1986

[SF91] G.S.Sohi and M Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors", Proc. 4th Int'l Conf. on Architectural Support for Programming Languages ad Operating Systems (ASPLOS-IV), pp53-62, 1991

[Sun89] Sun Microsystems, "The SPARC Architectural Manual, Version 8", Part No. 800-1399-09, 1989

[Zor92] G.Zorpette, "Technology 1992: Large Computers", IEEE Spectrum, Vol.29, No.1, pp33-35, 1992