Effectiveness of Register Preloading on CP-PACS Node Processor

Hiroshi Nakamura* Ken'ichi Itakura** Masazumi Matsubara**
Taisuke Boku** Kisaburo Nakazawa†

* Research Center for Advanced Science and Technology, University of Tokyo
** Institute of Information Sciences and Electronics, University of Tsukuba
† Department of Computer Science, University of Electro-Communications
{nakamura,itakura,matsu,taisuke,nakazawa}@arch.is.tsukuba.ac.jp

Abstract

CP-PACS is a massively parallel processor (MPP) for large scale scientific computations. On September 1996, CP-PACS equipped with 2048 processors began its operation at University of Tsukuba. At that time, CP-PACS was the fastest MPP in the world on LINPACK benchmark.

CP-PACS was designed to achieve very high performance in large scientific/engineering applications. It is well known that ordinary data cache is not effective in such applications because data size is much larger than cache size and because there is little temporal locality. Thus, a special mechanism for hiding long memory access latency is indispensable. Cache prefetching is a well-known technique for this purpose.

In addition to cache prefetching, CP-PACS node processors implement register preloading mechanism. This mechanism enables the processor to transfer required floating-point data directly (not via data cache) between main memory and floating-point registers in pipelined way.

We compare register preloading with cache prefetching by measuring real performance of CP-PACS processor and HP PA-8000 processor which implement cache prefetching and/or register preloading. The results revealed the following points. First, register preloading outperforms cache prefetching in consecutive data access. This is because cache prefetching suffers from lack of memory/cache throughput. Second, superiority of register preloading to cache prefetching significantly increases in stride access. This is because cache prefetching transfers not only the required data but also unnecessary data in the cache line. This leads to the waste of memory/cache throughput. Third, register preloading is also effective in indirectly indexed vector computations. Fourth, in multi-dimensional application, register preloading achieves stabler and better performance than cache-based computations even though blocking optimization is utilized in cache-based computations.

From these results, it is concluded that register preloading is superior to cache prefetching for large numerical applications.

1 Introduction

CP-PACS (Computational Physics, Parallel Array Computer System) [1] is a massively parallel processor dedicated for large scale scientific computations. On March 1996, CP-PACS with 1024 node processors began its operation at the Research Center for Computational Physics [2], University of Tsukuba, Japan. Then, on September 1996, CP-PACS was upgraded to 2048 processors. At that time, this machine was the fastest machine in LINPACK benchmark [3].

It is well known that ordinary data cache is not effective in large scientific applications because data size is much larger than cache size and because there is little temporal locality. It was reported in [4] that microprocessors equipped with ordinary data cache cannot achieve high performance in large scale applications. Therefore, some mechanisms of hiding long memory access latency are indispensable.

Cache prefetching is a well-known technique for hiding memory latency [5, 6]. This mechanism is implemented on some of recent advanced microprocessors, for example, Hewlett-Packard PA-8000 processor [7, 8].

However, cache prefetching has the following disadvantages.

- Traffic of data cache becomes twice as much as that of main memory if data access contains no temporal locality. This is because data cache is responsible for two kinds of data transfers. One is data transfer between main memory and cache. The other is data transfer between registers and cache.
- In stride or random data access, unnecessary data is transferred accompanied with the required data.
- Required data may be flushed out from data cache when other data is eventually fetched or prefetched into the same cache line.

Then, we adopted another mechanism called *register preloading* in addition to cache prefetching. In register preloading technique, requested data is transferred *directly* between main memory and registers in

0-8186-8424-0/98 $10.00

a pipelined way like vector processing. The different point from vector processing is that transfer of each data transfer is explicitly produced by the corresponding instruction. In this sense, our register preloading mechanism is scalar processing.

In [9, 10, 11], we reported only the simulation-based performance estimation of the node processor. In [1], real performance of the node processor was briefly presented, but performance comparison between register preloading and cache prefetching was not described. This paper focuses on the comparison of register preloading and cache prefetching based on real performance measurement.

2 CP-PACS Node Processor

This section briefly explains the register preloading mechanism of CP-PACS node processor. Details are described in [9].

2.1 Architecture

The node processor has the following characteristics in order to realize register preloading mechanism.

- larger number of floating-point registers
- direct data transfer between floating-point registers and main memory specified by instructions
- pipelined access to main memory

In spite of these enhancement, the node processor still keeps upward compatibility with PA-RISC 1.1 architecture[1] [12]. The architectural extensions are represented by slide-window structure of floating-point registers and newly introduced instructions.

(1) Slide-Windowed Floating-point Register Figure 1 shows the slide-windowed structure of floating-point registers.

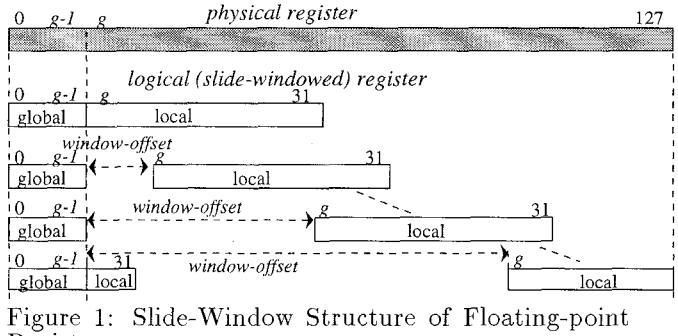

Figure 1: Slide-Window Structure of Floating-point Registers

The physical register space consists of logical windows. The number of total physical registers is 128. On the other hand, the number of registers in a logical window is 32, which is the same as what defined in PA-RISC 1.1 architecture. A logical window is divided into global part and local part. The number of registers in the global part, illustrated as g in Figure 1, is 8, 12, or 16, which is controlled by software.

Only one logical window is active at a certain time. Instructions of the original PA-RISC 1.1 architecture access only the registers within the active window. A pointer **FWSTP** (**F**loating-point **W**indow **ST**art **P**ointer) is introduced to identify the the location of the active window. This pointer always keeps the window-offset of the local part of the active window. The value of this pointer is changed by a new FWSTPset instruction mentioned below. By using this FWSTPset instruction, the active window acts as if it slides among the physical register space.

(2) New Instructions The following instructions are newly introduced.

- FWSTPset: Set the value of FWSTP and change the location of the active window.
- FRPreload: Load a floating-point data from memory into a floating-point register.
- FRPoststore: Store a floating-point data into memory from a floating-point register.

FRPreload and FRPoststore have the following characteristics, which distinguish these instructions from ordinary load or store instruction.

1. Data transfers between memory and registers caused by FRPreload/FRPoststore are non-blocking and executed in pipelined way.
2. On a cache miss, data is loaded/stored from/to main memory to/from a floating-point register *directly* without replacing any line of data cache.
3. FRPreload/FRPoststore can identify all the physical registers as the destination/source, regardless of the location of the active window.

Owing to the first characteristic, memory access latency is successfully hidden by issuing FRPreload instructions well in advance without disturbing other instructions' executions. That is, memory access latency is hidden by *register preloading*. The second point prevents the processor from performance degradation caused by insufficient data cache throughput.

In order to hide longer access latency, more registers are required. This is because FRPreload instructions should be issued earlier and because each FRPreload instruction reserves one destination register. Unless sufficient registers are provided, FRPreload instructions can not be scheduled enough earlier. Due to the third point, all the physical registers, that is 128 registers, can be utilized for this purpose. According to our previous simulation-based study [10], 128 registers are enough to hide memory latency of 100 cycles.

[1] Our node processor can keep upward compatibility with other load/store architectures. PA-RISC 1.1 is selected just as one candidate.

2.2 How to Tolerate Memory Latency by Register Preloading

We will explain how register preloading tolerates memory access latency by using an example of vector addition `C(i)=A(i)+B(i)`.

```
LOOP:  FRPreload        A(i+1)→fr30[+2]
       ADD              fr30+fr31 → fr31
       FRPreload        B(i+1) → fr31[+2]
       FRPoststore      fr31 → C(i)
       FWSTPset         FWSTP+2 → FWSTP
       Branch to LOOP
```

Figure 2: Vector Addition Code

Figure 2 shows the core part of the vector addition code. The prologue and the epilogue part are omitted from this figure.

The first instruction preloads `A(i+1)` into `fr30` of another window which is ahead of the current active window by 2 registers. The fifth instruction (`FWSTPset`) specifies that the active window is moved ahead by 2 registers. Thus, `fr30[+2]` of the current iteration is the same as `fr30` of the next iteration[2]. Therefore, `A(i+1)` preloaded into `fr30[+2]` is utilized by `ADD` instruction of the next iteration.

Suppose this code is executed on one-way issue pipelined processor. Then, 6 cycles are required for executing one iteration because one iteration consists of 6 instructions. Because preload of `A(i+1)` is requested by the first instruction `ADD` and because the data is used by the second instruction of the next iteration, `2-1+6 = 7` cycles of memory latency can be hidden for `A(i+1)`. By increasing the number of underlined part in Figure 2, longer memory latency is hidden. For example, suppose all the underlined parts are modified into `+4`. Then, the preloaded data is used in the *second* next iteration. As a result, `2-1+(6×2) = 13` cycles of memory latency can be hidden.

Qualitative Comparison of Register Preloading and Cache Prefetching Register preloading is essentially different from cache prefetching in the following points.

- The minimum data granularity in cache prefetching is the line size, which is usually larger than one floating-point data. Thus, in non-consecutive data accesses, unnecessary data in the same cache line must be transferred accompanied with the requested data. This wastes both memory and cache throughput. Register preloading does not suffer from this problem because only one floating-point data is transferred by one instruction.

- In cache prefetching, traffic of data cache becomes twice as much as that of main memory if data access contains no temporal locality. This is because cache must handle two kinds of data transfers. One is data transfer between main memory and cache. The other is data transfer between registers and cache. Thus, cache prefetching tends to suffer from lack of data cache throughput. Register preloading does not suffer from this problem because data is transferred directly between registers and main memory.

- The live range of preloaded data is completely under the control of software. Compiler can schedule instructions so that preloaded data certainly resides in the register until its usage. On the other hand, there is no guarantee that prefetched data resides in the cache until its usage, because other data may be eventually fetched or prefetched into the same cache line.

In order to tolerate long access latency by register preloading, more registers are required than usual. Indeed, CP-PACS node processor implements 128 floating-point registers. However, it is reported in [13] that implementation of 128 floating-point registers leads to only 5% increase of chip area.

3 Performance Comparison of Register Preloading and Cache Prefetching

This paper focuses on the comparison of register preloading and cache prefetching. The comparison is based on real performance measurement of CP-PACS node processor and Hewlett-Packard PA-8000 processor. Since CP-PACS node processor implements both register preloading and cache prefetching mechanisms, performance measurement of this processor is very helpful for the comparison of the two mechanisms. HP PA-8000 is also chosen for performance measurement because it implements cache prefetching mechanism aggressively and because it is based on similar architectures (PA-RISC 1.1 for CP-PACS and PA-RISC 2.0 for PA-8000),

3.1 Platforms

Overview of the two processors is presented. Here, we focus on the ability of floating-point operations, data cache, and main memory because performance is mainly affected by these factors.

CP-PACS Node Processor This is a 2-way issue superscalar processor whose clock frequency is 150MHz. Since two floating-point operations can be executed every cycle, theoretical peak performance is 300MFLOPS.

The block diagram of the processor is shown in Figure 3. This figure illustrates interconnections between registers, data caches, and main memory. The first-level 16KB D-cache are implemented on the processor chip, while the second-level 512KB D-cache are off-chip. All the caches are direct-mapped. Line size of the 1st D-cache is 32B, whereas that of the 2nd D-cache is 128B. The processor is equipped with 16-way interleaved main memory, whose capacity is 64MB and

[2]The code of Figure 2 uses only the local part of logical windows.

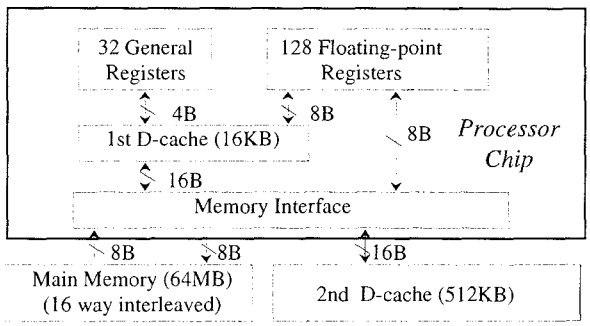

Figure 3: Block Diagram of CP-PACS Processor

maximum throughput is 8B/cycle (= 1.2GB/sec). Access latencies for the 1st and the 2nd cache are 1 cycle and 6 cycles respectively. Since main memory is an interleaved bank memory, access latency for main memory is not fixed but is longer than 50 cycles.

Both register preloading and software-based cache prefetching are implemented on CP-PACS node processor. Their specifications are summarized in Table 1.

D-Cache Status	Register Preloading	Cache Prefetching
1st hit	1st cache → register	no operations
1st miss 2nd hit	2nd cache → register, no replacement in 1st cache	2nd cache → 1st cache
1st miss 2nd miss	main memory → register, no replacement in 1st or 2nd cache	main memory → 1st&2nd cache

Table 1: Specification of Register Preloading and Cache Prefetching on CP-PACS processor

Hewlett-Packard PA-8000 We use a HP 9000 Model C160 workstation for the measurement. The CPU of this workstation is HP PA-8000 operating at 160MHz.

PA-8000 features a peak execution rate of four instructions per cycle. This processor implements fully-pipelined two floating-point multiply and accumulate (FMAC) units so that the peak floating-point throughput is four operations per cycle, which is twice as much as CP-PACS processor. The processor incorporates two complete load/store pipelines and dual-ported, single-level, off-chip, direct-mapped data cache whose capacity is 512KB and line size is 32B. The peak throughput of data cache 16B/cycle = 2.56GB/sec. The data cache has a latency of 2 cycles. Memory is connected via Runway bus which provides 768MB/sec throughput. Memory access latency is 50+ cycles. Software-based cache prefetching is available in this processor.

Summary of CP-PACS and PA-8000 Table 2 summarizes the characteristics of the two processors.

3.2 Benchmarks

In the evaluation, the following 4 kernels are used. Performance is measured with varying vector length.

		CP-PACS	PA-8000
Clock Frequency		150MHz	160MHz
Peak MFLOPS		300MFLOPS	640MFLOPS
Data cache	capacity	16KB(1st) 512KB(2nd)	512KB
	throughput	8B/cycle(1st)	16B/cycle
	line size	32B(1st) 128B(2nd)	32B
	latency	1 cycle(1st) 6 cycle(2nd)	2 cycle
Main Memory	throughput	1.2GB/sec (16way bank)	768MB/sec (Runway bus)
	latency	50+ cycle	50+ cycle
Cache Prefetching		available	available
Register Preloading		available	not available

Table 2: Summary of CP-PACS and PA-8000

Vector `A,B,C` represent double-precision floating-point data whereas vector `L` represents integer data.

- ADD `[C(i) = A(i) + B(i)]` : All the data accesses are consecutive.
- ADD5 `[C(i) = A(i*5) + B(i)]` : Vector `A` is accessed in stride of 5.
- RANDOM-SUMUP `[s = s + A(L(i))]` : Vector `A` is accessed randomly with sparsity of 10%.
- MATRIX-MULTIPLY `[C(i,j) = A(i,k) * B(k,j)]`

Using simple kernels is helpful for making clear the characteristics and discussing the benefit of register preloading and cache prefetching. MATRIX-MULTIPLY is selected in order to compare register preloading technique and cache-based processing which uses blocking optimization

Compilation These four kernels are complied using the following three optimization.

base Neither cache prefetching nor register preloading is enabled.

prefetch Only cache prefetching is enabled.

preload Only register preloading is enabled except for RANDOM-SUMUP. This optimization is available only for CP-PACS processor. In RANDOM-SUMUP, both register preloading and cache prefetching are enabled.

On CP-PACS processor, Fortran compiler 'f90' is used, whereas C compiler 'cc' is used on PA-8000. Since all the benchmarks are simple kernels, this does not affect the optimization quality, which is confirmed from the generated codes. Table 3 shows the flags used for each optimization. In all the optimization on both platforms, the innermost loops are unrolled four times.

Opt.	CP-PACS	PA-8000
base	nopvec†, opt(o(s),fold(2),prefetch(0))	+O2
prefetch	nopvec†, opt(o(s),fold(2),prefetch(1))	+O2, +Odataprefetch
preload	pvec†, opt(o(s),fold(2),prefetch(0)‡)	*not available*

† pvec(nopvec) indicates (not) using register preloading
‡ prefetch(1) is used for RANDOM-SUMUP instead

Table 3: Flags for Compilation

In RANDOM-SUMUP, cache prefetching is used together in preload optimization in order to hide latency for integer data. That is, register preloading is used for hiding the latency of floating-point data `A`, whereas cache prefetching is used for hiding the latency of integer data `L` in register preloading optimization. CP-PACS processor does not implement register preloading mechanisms for integer data. Note that cache prefetching is used for hiding the latency of all the data in cache prefetching optimization.

As for MATRIX-MULTIPLY, the loop-order is changed depending on the optimizations. It is well known that ikj-loop [Figure 4(a)] is the best for cache-based RISC processors whereas ijk-loop [Figure 4(b)] is the best for vector processors. Therefore, ikj-loop is used for base and prefetch, whereas ijk-loop is used for preload. Register preloading is similar to vector processing. In practice, ikj-loop and ijk-loop achieved the best performance on each optimization.

```
for (i=0; i< ARYSIZE; i++){
  for (k=0; k< ARYSIZE; k++){
    reg1 = a[i][k];
    for (j=0; j< ARYSIZE; j++){
      c[i][j] += reg1*b[k][j]; } } }
```

(a): ikj-loop

```
for (i=0;i < ARYSIZE; i++){
  for (j=0; j< ARYSIZE; j++){
    for (k=0; k< ARYSIZE; k++){
      reg1 += a[i][k]*b[k][j]; }
    c[i][j] = reg1;
    reg1 = 0; } }
```

(b): ijk-loop

Figure 4: Loop-Order of MATRIX-MULTIPLY in C

Cache Warming-Up In order to avoid disturbance from instruction cache, each kernel is executed several times repeatedly. We measured the required time for each execution, and selected the shortest time. Under this measurement, instruction cache is warm. If the vector length is small and data size is smaller than cache size, data cache is also warm in base and prefetch optimization. Note that data cache is still cold in preload optimization because FRPreload or FRPoststore operation itself does not transfer any data into data cache.

3.3 Results and Discussions

(1) ADD `[C(i) = A(i) + B(i)]`: Figure 5 shows the relation between performance and vector length. The following points are observed.

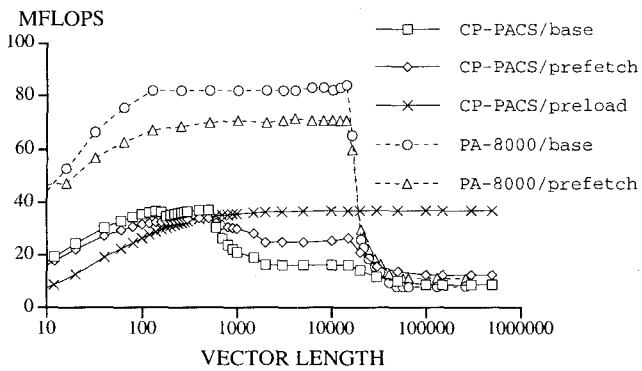

Figure 5: Performance of ADD LOOP

base vs. prefetch

- For short vectors, prefetch is slower than base. This is because useless prefetch instructions are inserted.
- When the vector length is between 600 and 20000, 1st D-cache misses but 2nd D-cache hits on CP-PACS. In this case, CP-PACS/prefetch is better than CP-PACS/base.
- When the vector length is larger than 50000, prefetch is faster than base both on CP-PACS and PA-8000. However, the superiority is not significant. For example, at the vector length of 100000, CP-PACS/prefetch is 44% faster than CP-PACS/base and PA-8000/prefetch is 39% faster than PA-8000/base.

prefetch vs. preload

- When vector length is short, CP-PACS/preload is slower than CP-PACS/prefetch or CP-PACS/base. This is because 1st D-cache hits for base and prefetch, whereas 1st D-cache does not hit for preload.
- Surprisingly, performance of CP-PACS/preload does not degrade for long vectors. CP-PACS/preload at long vector achieves almost the same performance as CP-PACS/base at short vector where 1st D-cache always hit. This indicates that preload can successfully hide the memory latency at long vectors.
- Although PA-8000's ability of floating-point computations is about 2 times higher than CP-PACS,

PA-8000/prefetch achieves only 30% performance of CP-PACS/preload when the vector length is 100000 or more. This represents that register preloading is more effective than cache prefetching for long vectors.

Here, we will further explain the reason why performance of CP-PACS/prefetch degrades in increasing vector length. The reason is insufficient throughput of memory and cache.

Suppose the load of consecutive 16 double-precision floating-point data (=128B). In cache prefetching, the corresponding code consists of 16 load instructions and 4 prefetch instructions because the line size of the 1st cache is 32B. Here, we estimate the required time for prefetching.

- If the data exists in the 1st cache, it takes 4 cycles. This is the required time for issuing 4 prefetch instructions.
- If the data exists not in the 1st cache but in the 2nd cache, it takes 8 cycles. This is because throughput from the 1st cache to the 2nd cache is 16B/cycle and amount of transferred data is 128B. Thus, additional 4 cycles are required.
- If the data does not exist in 1st or 2nd cache, this execution requires 22 cycles. The first prefetch instruction causes 128B data transfer from main memory into the 2nd cache[3]. This transfer takes 16 cycles because memory throughput is 8B/cycle. For the other three prefetch instructions, 2nd cache hits. Thus, these prefetch instructions bring about 32B data transfer from the 2nd cache into the 1st cache, which takes 2 cycles. In total, 16+2×3=22 cycles are required[4].

In this way, performance of cache prefetching degrades due to the lack of the cache and memory throughput. The 2nd cache takes a middle position in the memory hierarchy (memory ↔ 2nd-cache ↔ 1st-cache). This observation also indicates that data traffic of the middle memory gets very heavy. This is because the middle memory should provide data transfers to/from both higher and lower memory. Similar reasons hold for performance degradation of PA-8000/prefetch, we suppose.

On the contrary, in CP-PACS/preload, only 16 cycles are taken for transferring 128B data from main memory into registers[5]. Register preloading does not suffer from the above problem because data caches are bypassed.

(2) ADD5 `[C(i) = A(i*5) + B(i)]`: This kernel is similar to ADD but contains stride data access. The result is shown in Figure 6. The following is observed.

[3]During this transfer, 32B data is transferred into the 1st cache simultaneously.

[4]Real performance degrades further because the degree of non-blocking is limited. CP-PACS allows up to 4 outstanding misses, whereas PA-8000 allows up to 10 outstanding misses [7].

[5]Please remind that the above estimation of CP-PACS/prefetch does not include the required time for 16 load instructions.

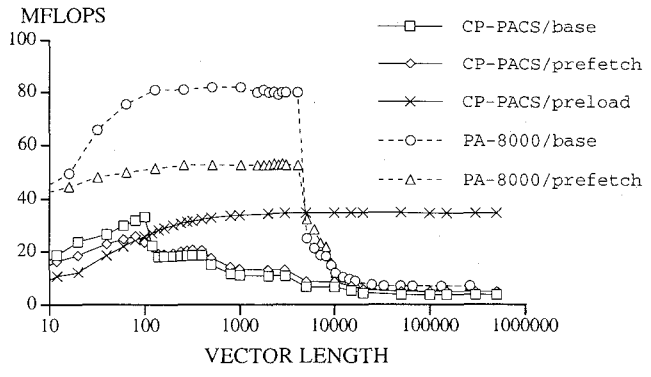

Figure 6: Performance of ADD5 Loop

- Compared with Figure 5, when vector length is short and data size is smaller than cache capacity, performance of prefetch gets worse. This is because in stride data accesses, more prefetch instructions are inserted into the object codes.
- Comparing with Figure 5, performance of prefetch is worse for long vectors. For example, at the vector length of 100000, CP-PACS/prefetch achieves only 40% and PA-8000-prefetch achieves only 63% performance of ADD kernel.
- On the contrary, CP-PACS/preload achieves almost the same performance as in ADD kernel (Figure 5).
- Although PA-8000 is about 2 times faster than CP-PACS in potential, CP-PACS/preload is almost 5 times faster than PA-8000/prefetch when the vector length is 100000 or more.

The first and second observations reveal that cache prefetching is not very effective in stride data access. In stride data access, unnecessary data is prefetched accompanied with the requested data, which leads to the waste of memory throughput. Therefore, cache prefetching heavily suffers from lack of memory/cache throughput.

On the contrary, the third and the fourth points show that *register preloading* is very effective even in stride data access. The essential reason for this good result is that the minimum data granularity handled by register preloading is one floating-point data, not cache line size. Therefore, unnecessary data transfer does not occur.

(3) RANDOM-SUMUP `[s = s + A(L(i))]`: The result is shown in Figure 7. The following is observed.

- Performance of prefetch is almost the same as base on both CP-PACS and PA-8000.
- Preload significantly outperforms prefetch at long vectors. For example, when the vector length is 200000, CP-PACS/preload is 10 times faster than

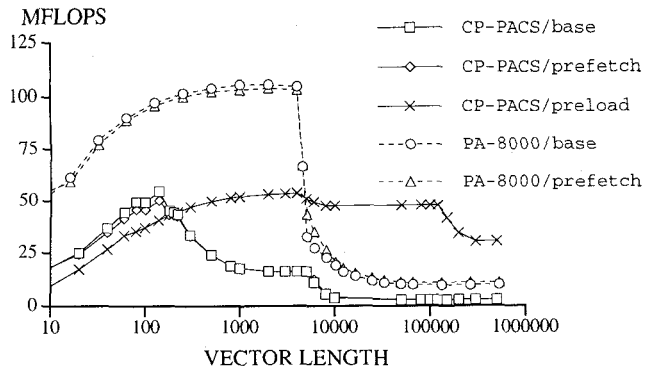

Figure 7: Performance of RANDOM-SUMUP Loop

CP-PACS/prefetch. CP-PACS/preload is also 2.8 times faster than PA-8000/prefetch although PA-8000 has higher computation ability.

- Performance of CP-PACS/preload degrades twice in increasing vector length.

The first point reveals that cache prefetching is not effective in indirectly indexed vector computations. On the contrary, the second point represents that register preloading is very effective also in indirectly index vector computations. The reason for this contrast is that latency for indirectly indexed vector `A` is hidden by register preloading but not hidden by cache prefetching.

Cache prefetching can easily hide latency for index vector `L`. In the compiled code, `prefetch L(i+`*prefetch-distance*`)` and `load L(i)` are executed in the same iteration. Since `L` is accessed regularly, *prefetch-distance* is constant and given as an immediate value in the instruction. However, prefetching `A` is very hard because `A` is accessed in irregular. Then, prefetch request of `A(L(i))` and load request of `A(L(i))` must use the same index register which holds the offset value of `L(i)`. Since time period between these two requests is quite long (50+ cycles for CP-PACS and PA-8000), the value of base/index register cannot be kept in a register. Indeed, `A(L(i))` is not prefetched in the compiled code on CP-PACS or PA-8000 (the compilation option of Table 3 is used). Therefore, performance of prefetch degrades seriously caused by long access latency for indirectly index vector `A`. On the other hand, in register preloading, `A` is easily preloaded into a register because only one FRpreload instruction is required for transferring `A(L(i))`. We need not keep the value of base/index register for long time. We presented in [11] the method of hiding memory access latency in indirectly indexed vector by register preloading in more detail.

The third point is explained as follows. In CP-PACS/preload, latency for floating-point data `A` is hidden by register preloading whereas latency for integer data `L` is hidden by cache prefetching[6]. Therefore, when the working set size of vector 'L' gets larger than 1st or 2nd D-cache, performance degrades.

(4)
MATRIX-MULTIPLY `[C(i,j) = A(i,k) * B(k,j)]`:
The results is shown in Figure 8. Blocking optimization is applied to base and prefetch. The block size is fixed to 32×32 for CP-PACS and 128×128 for PA-8000. The selected matrix-sizes are 100, 200, 256, 300, 400, 500, 512, 600, 700, 768, 800, 900, 1000, and 1024.

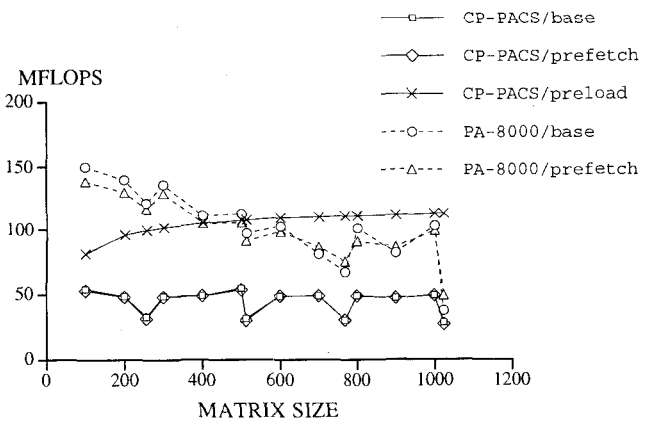

Figure 8: Performance of MATRIX-MULTIPLY

- Performance of base and prefetch degrade at certain matrix sizes (for example, 768 and 1024). This is because line conflicts occur in data cache.
- On the contrary, performance of preload is stable and independent from matrix size.
- Performance of prefetch is slightly worse than base on PA-8000 in most cases.
- On PA-8000, even though blocking optimization is applied, performance of base or prefetch degrades when the matrix size gets larger than cache size (note that 512KB = 256×256 matrix size). For example, PA-8000/base at 800×800 matrix achieves about 67 % performance of 100×100 matrix.
- CP-PACS/preload achieves almost the same or slightly better performance than PA-8000/base, although PA-8000 has about two times higher computation power than CP-PACS. This observation holds even if excluding the particular bad points for PA-8000 (768 and 1024).

Comparison of the first and the second point illustrates that register preloading is easier for performance optimization. The first problem can be solved

[6]Register preloading is not available for integer data on CP-PACS.

to some extent by selecting adequate block size [14] or by inserting dummy data [15]. However, the optimal block size or dummy data size sensitively depends on matrix size and cache parameters including capacity, line size, number of associativity and so on. On the contrary, in register preloading, stable and good performance is achieved by using same source code.

The third point indicates that cache hit ratio of base is quite high in this computation. In other words, blocking optimization is very effective in this computation.

The forth and the fifth points reveal that register preloading is still better than cache-based computations even though very effective blocking optimization is incorporated into cache-based computations.

4 Conclusion

In large scale scientific computations, how to hide long memory access latency is the key point for achieving high-performance. We have proposed a new technique, *register preloading*, as the technique of latency hiding. Register preloading is implemented on our CP-PACS node processors.

In this paper, performance evaluation of register preloading and cache prefetching is presented. The evaluation is based on real performance measurement of CP-PACS node processor and HP PA-8000 processor. The results revealed the following points.

- Register preloading is superior to cache prefetching for long consecutive vectors. This is because cache prefetching suffers from lack of memory/cache throughput.
- In stride data access, superiority of register preloading to cache prefetching increases significantly. This is because unnecessary data is prefetched in cache prefetching, which leads to the waste of memory/cache throughput.
- Register preloading is also effective in indirectly indexed vector computations.
- In multi-dimensional application, register preloading achieves stabler and better performance than cache-based computations even though blocking optimization is incorporated into cache-based computations.

From these results, it is concluded that register preloading is superior to cache prefetching for large numerical applications.

References

[1] T. Boku, K. Itakura, H. Nakamura and K. Nakazawa, *CP-PACS: A massively parallel processor for large scale scientific calculations*, Proc. of ACM Int'l Conf. on Supercomputing '97, 1997

[2] `http://www.rccp.tsukuba.ac.jp`

[3] J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software*, `http://www.netlib.org/benchmark/performance.ps`, as of October, 20, 1996

[4] 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, pages 132-141, 1990

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

[6] T. Chen and J.L. Baer, *A Performance Study of Software and Hardware Data Prefetching Scheme*, Proc. 21th Int'l Symp. on Computer Architecture, pages 223-232, 1994

[7] V. Santhanam, E. Gornish, and W. Hsu, *Data Prefetching on the HP PA-8000*, Proc. 24th Int'l Symp. on Computer Architecture, pages 264-273, 1997

[8] A. Scott, K. Burkhart, A. Kumar, R. Blumberg, and G. Ranson, *Four-way Superscalar PA-RISC Processors*, Hewlett-Packard Journal, Vol.48, No.4, 1997

[9] H. Nakamura, H. Imori, K. Nakazawa, T. Boku, I. Nakata, Y. Yamashita, H. Wada, and Y.Inagami, *A Scalar Architecture for Pseudo Vector Processing based on Slide-Windowed Registers*, Proc. of ACM Int'l Conf. on Supercomputing '93, pages 298-307, 1993

[10] H. Nakamura, K. Nakazawa, H. Li, H. Imori, T. Boku, I. Nakata, and Y. Yamashita, *Evaluation of Pseudo Vector Processor based on Slide-Windowed Registers*, Proc. of 27th IEEE Hawaii Int'l Conference on System Sciences-27, pages 368-377, 1994

[11] H. Nakamura, T. Wakabayashi, K. Nakazawa, T. Boku, H. Wada, and Y. Inagami, *Pseudo Vector Processor for High-Speed List Vector Computation with Hiding Memory Access Latency*, Proc. of 1994 IEEE TENCON, pages 338-342, 1994

[12] Hewlett-Packard Company, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual (Third Edition)*, Manual Part Number 09740-90039, 1994

[13] K. Shimamura, S. Tanaka, T. Shimomura, T. Hotta, E. Kamada, H. Sawamoto, T. Shimizu, and K. Nakazawa, *A Superscalar RISC Processor with Pseudo Vector Processing Feature*, Proc. of IEEE ICCD'95, pages 102-109, 1995

[14] M. Lam, E. Rothberg, and M. Wolf, *The Cache Performance and Optimization of Blocked Algorithms*, Proc. of ASPLOS-IV, pages 63-74, 1991

[15] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau, *A Data Alignment Techinique for Improving Cache Performance*, Proc. of IEEE ICCD'97, pages 587-592, 1997