Efficient Software Data Prefetching for a Loop with Large Arrays *

Se-Jin Hwang and Myong-Soon Park

Department of Computer Science
Korea University
Seoul, Korea, 136-701
{hsj,myongsp}@cslab1.korea.ac.kr

Abstract. In this paper, we propose a software data prefetching mechanism to cope with following two unfavorable phenomenons from large arrays. One is the failure of the reuse, and the other is the effect of the presence of unnecessary prefetching instructions. Also, we realized the proposed mechanism into a preprocessor,*LOOP*.

1 Introduction

The memory latency problem by cache miss is very serious in superscalar microprocessor that allows the simultaneous issuing of multiple instructions[2]. Therefore, techniques to reduce or tolerate large memory latency become essential for achieving high processor utilization. As one of the techniques to mitigate the large memory latency, the researches on data prefetching have been done for a few years[1, 2, 3, 4].

Prefetching transaction can be triggered either by a dedicated instruction[1, 4] or by a hardware auto-detection[3]. We call the former as software data prefetching, and the latter hardware data prefetching. The major drawback of software data prefetching is extra processor cycles to execute prefetching instructions inserted by a compiler[4]. Henceforth, the effectiveness of a software scheme is sensitive to how well the compiler inserts these instructions.

Mowry *et al* proposed a selective prefetching mechanism[4] which inserts prefetching instructions for array elements likely to show cache miss. The reuse analysis by the compiler makes it possible that we should insert only necessary prefetching instructions. In this mechanism, it is very crucial that the compiler should insert *necessary prefetching instructions* and transform a loop to exploit *data reuse*. However, these two items can be threatend by large arrays.

Unless a cache could hold one row of array accessed non-sequentially, some elements might be displaced from a cache[5]. Such fact urges us to load again data that used to be in the cache.

In addition, larger the size of an array becomes, more the number of prefetch instructions must be executed. When the array has LCD[2], same elements of the

* This paper was supported (in part) by NON DIRECTED RESEARCH FUND, Korea Research Foundation

[2] We abbreviate loop carried dependency to 'LCD' in the rest of this paper.

array may be used at different loop iterations. Such fact may turn out many prefetching instructions to be unnecessary, if they were inserted uncarefully. To make matters worse, provided that the array were large, the amount of wasted cycles to execute the unnecessary prefetching instructions could be non-negligible.

In this paper, we propose software data prefetching mechanism which considers LCD, and the size of a cache along with the number of data that will be referenced. The proposed mechanism has two philosophies to cope with the failure of the data reuse: One is to reuse it desparately, and the other to give it up. Also, in order to eliminate unnecessary prefetching instructions for an array with LCD, it apropriately transforms a loop based on an accurate analysis.

The organization of the rest of this paper is as follows. First, section 2 discribes the motivation of this work. Section 3 proposes software data prefetching mechanism to mitigate the problem indicated in section 2. Section 4 describes the simulation method to measure the execution cycles of loops transformed by *LOOP*[6]. *LOOP* is a preprocessor that transforms a loop in order to insert prefetching instructions by using proposed scheme. We finally conclude in section 5.

2 Motivation

In this section, we address the types of performance degradation by large arrays in a loop.

2.1 The failure of data reuse by large arrays

When the non-sequentially accessed array is so large that a cache cannot hold one row of the array, some element could be displaced before reuse[5]. Therefore, it is necessary that we should reload data that used to be in the cache.

To estimate how successfully array elements can be reused as the array size increases, we transformed VPENTA[3] using the selective prefetching mechanism[4]. VPENTA is a nested loop with depth 2, and has eight array variables within a body. Furthermore, since all arrays in VPENTA are accessed non-sequentially, it is heavily sensitive to the size of the arrays. When we applied such selective mechanism into VPENTA, we assumed that the size of cache line is twice as much as that of one array element. Two innermost loops are generated after transforming VPENTA with the selective mechanism. The first one includes prefetching instructions, but the second does not, in order to use elements that have been brought into a cache during the execution of the first innermost loop.

We estimated the hit ratio of accessing array Y in the second innermost loop in VPENTA varying the size of the array. Graphs in Figure 1 present how well the array Y can be reused with increasing the size of it. In general, the graph for the size of 64×64 shows higher hit ratio than that for the size of 256×256.

It indicates that the larger the size of array is, the fewer the possiblity of reusing data accessed non-sequentially is. This phenomenon means that more

[3] VPENTA is a benchmark loop included in NAS kernels

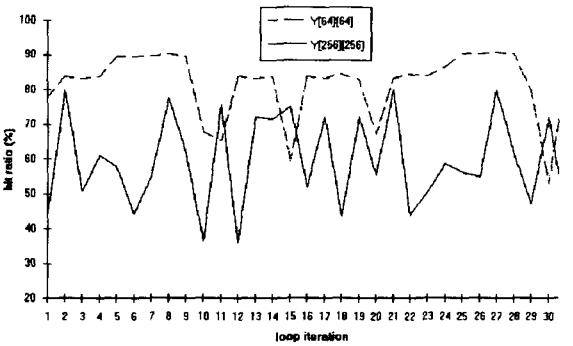

Fig. 1. Degree of reusability of array Y in VPENTA

reloading costs must be paid as the number of accessed array elements is increased.

To remove such overhead, our proposed mechanism examines whether array elements will be held in a cache or not. Unless we can reuse them in the cache, it applies indiscriminate prefetching mechanism[1] proposed by Porterfield. Otherwise, we applies the selective mechanism.

2.2 The negative effect of large arrays with LCD

Unnecessary prefetching instructions for an array with LCD might be easily inserted into a loop. Unfortunately, when the array is large, the total waste cycles to execute such instructions comes to be considerable. Therefore, it is essential that we consider prudently LCD to achieve efficient software data prefetching.

To illustrate how many prefetching instructions are turned out to be unnecessary by LCD, we present Figure 2. When we insert the prefetching instruction, we assume that the size of cache block is twice as much as that of an element of the array A.

```
for(i=0;i < 100;i++){
    prefetch(&(A[i]));
    A[i + 1] = A[i] + ...;
    }
```

(a) a prefetching instruction for $A[i]$ is inserted

```
for(i=0;i<100;i++){
    prefetch(&(A[i + 1]));
    A[i + 1] = A[i] + ...;
    }
```

(b) a prefetching instruction for $A[i + 1]$ is inserted

Fig. 2. Two cases of a loop which includes a prefetching instruction for an array A with LCD

Under this assumption, we can presume that the percentage of executed unnecessary prefetching instructions will be 99% at the execution of the loop in

Figure 2(a). All executed prefetching instructions, except for the first, will be unnecessary, since the object of the prefetching has been already cached before one iteration.

In case the argument of the prefetching instruction is substituted with '&(A[i+1])' as in Figure 2(b), the percentage of unnecessary prefetching instructions comes to be about 50% with the same presumption described above. At every two iterations, the inserted prefetching instruction tries to access array element which has been already cached in.

From Figure 2, we can know that LCD is should be taken into account on inserting prefetchinging instructions. Our proposed mechanism transforms properly a loop with consideration of LCD of an array. To avoid inserting unnecessary prefetching instructions, it peels as well as unrolls the loop.

3 Proposed Software Data Prefetching

In this section, we propose a software data prefetching mechanism to cope with the problems indicated in section 2. First, in subsection 3.1, we describes memory reference pattern of an array within a loop. From subsection 3.2 to 3.4, we propose loop transformation algorithms to insert efficiently prefetching instructions into the loop, according to the memory reference pattern. We put these algorithms all together to be applicable to general nested loop in subsection 3.5.

3.1 Memory reference pattern

We denote the k-th loop nest in nested loop by L_k ($1 \leq k \leq n$, n : the depth of nested loop). As the level of a loop is proceeded one by one from the innermost to the outermost, we assume that the value of k increases by one. I_k also refers to the loop control variable of the L_k. We use $Loc(I_k)$ as a notation to express the corresponding position of loop control variable I_k into subscripts of an array. ${P^a}_k$ stands for the memory reference pattern that comes out at running an iteration of L_k. Since memory reference pattern is intrinsically the sequence of repeated cache hit and miss, we can describe P_k as $(M^a\ H^b)$, here, M and H mean cache miss and hit respectively. a and b refers to the number of repetition of cache miss and hit respectively. F_k also represents the upper bound of L_k.

Memory reference pattern can be classified by comparing $Loc(I_k)$ with $Loc(I_{k-1})$ of all array variables into a loop body : *sequential*, *non-sequential*, *fixed-positional* reference patterns. As the corresponding position of I_k into array subscripts is moving from the right to the left, we assume that $Loc(I_k)$ is increasing by one. In case that I_k corresponds to no position of the array subscripts, $Loc(I_k)$ is 0. We describe the features of memory reference patterns, and the corresponding result of comparing $Loc(I_k)$ with $Loc(I_{k-1})$ as follows.

- Sequential memory reference pattern
  The adjacent array elements can be reused only a few iterations later.

$$Loc(I_1) = 1\ ,\ (k = 1) \tag{1}$$

$$Loc(I_k)\ >\ Loc(I_{k-1})\ ,\ (k > 1) \tag{2}$$

- Non-sequential memory reference pattern
  The adjacent array elements can be reused after one row of the array was entirely accessed.

$$Loc(I_1) \neq 1 \ , \ (k = 1) \tag{3}$$

$$Loc(I_k) \ < \ Loc(I_{k-1}) \ , \ (k > 1) \tag{4}$$

- Fixed-positional memory reference pattern
  This array element comes to be reused, after the first compulsory miss.

$$Loc(I_k) \ = \ 0 \tag{5}$$

From subsection 3.2 to 3.4, three loop transformation algorithms are introduced. They insert efficiently prefetching instructions into a loop, according to the correspoding memory reference pattern.

3.2 Loop transformation algorithm for sequential reference pattern

When we access an array sequentially in a loop, we come to reference the adjacent elements in a memory. Therefore, more than one cache hit will occur, after one cache miss. It is the sequential reference pattern. This sequence will be repeated during our references to this array. ${P^a}_k$, sequential access pattern of an array a can be represented at k-th loop nest as follows.

$${P^a}_1 = (\ M \ H^v \ )^{F_1/(v+1)} \ , \ (k = 1) \tag{6}$$

$${P^a}_k = ({P^a}_{k-1})^{F_k/(v+1)} \ , \ (k > 1) \tag{7}$$

Above, v can be calculated by

$$v \ = \ \frac{array \ element \ size}{cache \ line \ size} - 1. \tag{8}$$

Since the memory reference pattern of the lower level is preserved as shown in the above expression (7), transformation algorithm only have to work at the innermost loop level.

When an array has LCD, many references to the array exist within a loop body. Therefore, we can express the memory reference pattern of this array at k-th loop nest as ${P^a}^n{}_k, (n > 0)$. Here, n means the number of memory references which access the array a within a loop body. Sequential reference pattern of the array a with LCD can be expressed like the following.

$${P^a}^n{}_k = (\ {P^a}_k \ H^x \ ) \ , \ (x \geq 0) \tag{9}$$

We can interpret the expression (9) that a reference to an array with LCD tends to access an array without LCD for a while, but, it always comes to show cache hit after the particular loop iteration. From such particular loop iteration, this reference uses array elements which have been already in the cache by other references with larger subscript. The reference with largest subscript accesses the array as if it had no LCD. Therefore, the value of x in expression (9) of such reference is 0.

Based on expression (6),(7) and (9), we present an algorithm for an array with sequential reference pattern in Figure 3.

```
if( LCD appears ){
  unrolling = the smallest x in expression (9).
  /* unrolling indicates how many iterations will be peeled. */
  peel iterations as much unrolling.
}
if( k = 1 ){
  /* k indicates the level of a nested loop */

  unroll the k-th loop v + 1 times.
  insert prefetching instructions in the body of unrolled loop.
  split unrolled loop.
  /* we split the loop into prologue, body
  and epilogue loops, to secure enough prefetch distance. */
}
```

Fig. 3. Loop transformation algorithm for an array with sequential reference pattern

3.3 Loop transformation algorithm for non-sequential reference pattern

Non-sequential reference pattern can be represented as follows.

$$P^a{}_1 = (M^{F_1}\ H^{F_1 \times v})\ ,\ (k = 1) \tag{10}$$

$$P^a{}_k = ((P^a{}_{k-1}\{M\})^{F_k}\ (P^a{}_{k-1}\{H\})^{F_k \times v})\ ,\ (k > 1) \tag{11}$$

Here, $P^a{}_{k-1}\{M\}$ and $P^a{}_{k-1}\{H\}$ represent a sequence of cache hit and a sequence of cache miss of memory reference pattern at L_{k-1} respectively. v can be calculated using expression (8). As we can see from the expression (11), the sequences of cache hit and miss shown at the execution of the $(k-1)$th loop are preserved at k-th loop nest. Therefore, we only have to unroll the k-th loop v times. After unrolling, our algorithm for this pattern inserts prefetching instructions only into the first $(k-1)$th loop. If the number of array elements that are referenced during the excution of k-th loop is so large that the cache cannot hold all referenced elements, some of these will conflict with old cached ones. Such conflicts among array elements prevent us from analyzing which elements can be reused. By examining the expression (12), we can determine whether such conflicts will occur or not.

$$F_k \times na \geq CS \times 2. \tag{12}$$

Here, na means the number of array elements that will be referenced, and CS the cache size. If the expression (12) is satisfied, memory reference pattern will be the followings.

$$P^a{}_1 = (M^{F_1})\ ,\ (k = 1) \tag{13}$$

$$P^a{}_k = P^a{}_{k-1}{}^{F_k}\ ,\ (k > 1) \tag{14}$$

We present the loop transformation algorithm for non-sequential reference pattern in Figure 4. It considers the number of array elements that will be referenced, and meet with the case that we can hardly exploit the reuse property of the cache. The algorithm in Figure 4 has two opposite philosophies: one is to ignore the reuse property, and the other to exploit it.

```
if( we can hardly reuse array elements in a cache ){
   /* the condition can be examined using expression (12) */
   inserts prefetching instructions for all arrays accessed at the k-th loop nest.
   /* Here, we ignore the reuse property in the cache,
   and, prefetch array elements indiscriminately */
}else{
   unrolls the body of k-th loop nest.
   inserts prefetching instructions only the first (k − 1)-th loop.
   /* Here, we unroll the loop based on the pattern
   of expressions (10),(11) to exploit reuse property */
}
```

Fig. 4. Loop transformation algorithm for an array with non-sequential reference pattern

3.4 Loop transformation algorithm for fixed-positional reference pattern

If an array does not include I_k as a subscript, this will be referenced like a single variable at the execution of k-th loop. This array is not affected by the decrement or increment of the value of the I_k. So, all accesses to this element will show cache hit continuously after one cache miss at the first time. We refer to such reference pattern as fixed-positional reference pattern. It can be represented as follows.

$$P^a{}_1 = (M\ H^{F_1-1})\ ,\ (k = 1) \tag{15}$$

$$P^a{}_k = (P^a{}_{k-1}\ H^{F_k-1})\ ,\ (k > 1) \tag{16}$$

Since only one cache miss is shown in expression (15), peeling one iteration is enough measure to optimize the number of prefetching instructions. We express it as an algorithm in Figure 5.

3.5 Generalized Algorithm

Algorithms from Figure 3 to Figure 5 can be applicable to only one loop level, and to only one memory reference pattern. It is practically essential that we should

peel the body of the k-th loop.
if($k = 1$) inserts prefetching instructions above the loop.

Fig. 5. Loop transformation algorithm for an array with fixed-positional reference pattern

```
k ← 1. /* the innermost loop level */
while( k ≤ the outermost loop level )
    P_s ← memory reference patterns shown at level k.
    /* P_s : temporary buffer holding reference patterns. */
    if( P_s includes sequential memory reference pattern )
            apply the algorithm in Figure 3.
    if( P_s includes non-sequential memory reference pattern )
            apply the algorithm in Figure 4.
    if( P_s includes fixed-positional memory reference pattern )
            apply the algorithm in Figure 5.
    k ← k + 1. /* navigates to the upper loop nest */
}
```

Fig. 6. Generalized loop transformation algorithm

make these algortihms cooperate one another to transform a general nested loop and to insert prefetching instructions into it.

Therefore, we generalize three algorithms to insert prefetching instructions efficiently into a nested loop. Figure 6 shows a generalized loop transformation algorithm. It navigates every loop nests from the innermost to the outermost. It identifies memory reference patterns of all arrays at each loop nest, and then transforms a loop using a corresponding algorithm.

4 Simulation

In this section, we describe a simualtion method in order to measure the execution cycles of loops transformed by *LOOP*[6]. We also present our simulation results.

4.1 Environments

We defined our own $mutated - C$ language[6]. Our $mutated - C$ includes only *for*, *if* and *assignment* statements.

The preprocessor, *LOOP*, can understand only such restricted specifications of $mutated - C$. *LOOP* transforms '*for*' statements in $mutated - C$, and transforms it with prefetching instructions. The prefetching instructions inserted into the loop like an external function call, for example, '$prefetch(\&(A[i][j]))$;'. The

argument of this function is the address of one array element. The loop produced from the preprocessor takes a form of the program written in an original C language.

We obtain machine codes of the transformed loop using *dlxcc*, the compiler in DLX simulator. After compilation with *dlxcc*, the prefetching instructions inserted by *LOOP* are changed into DLX machine code, *jal*. However, since DLX simulator cannot support our requests, such as a prefetching request buffer, we made our own simulator using C.

To execute the DLX machine codes in our simulator, it is necessary that we should change every DLX machine codes into functions defined in our simulator. However, it was so laborious and monotonous work that we made a converter called *DLX2C*[6] that performs the dirty work only in a few seconds.

DLX2C produces a C program that describes the DLX machine instructions. After linking it with our simulator, we can obtain an executable file. This executable file keeps track of the machine codes of the loop transformed by *LOOP*, and calculates execution cycle, processor idle cycle etc. To quantify the execution cycle, we assume that the cycle time spent at the execution stage of individual instructions of DLX as Table 1.

Table 1. Assumed instruction cycle

DLX instructions	cycles
add addi addui sub subi movfp2i movi2fp lhi sll jal j	*1*
sleu slt sle sgtu sge sgt bnez beqz	*2*
multf divf	*3*

Benchmarks evaluated in this paper are a part of Lawrence Livermore Loop and VPENTA. LLLs of number 1, 7, 9, 11, 18 and 21 are used. LLL 1, 7, 11, 18 loop have arrays with sequential reference pattern. LLL 9 has only one array variable with non-sequential reference pattern. We set the size of arrays in these loops to be 1000.

All arrays accessed in VPENTA have only the non-sequential reference pattern, whereas, all arrays in LLL 21 have three kinds of reference pattern. On simulating VPENTA, we vary the size of array variables in it to 64×64, 128×128, 256×256 and 512×512. We also vary the total iteration number of LLL 21 to the followings: $20 \times 20 \times 100$, $20 \times 20 \times 500$, $20 \times 20 \times 1000$, and $20 \times 20 \times 2000$. Since cache pollution phenomenon breaks out seriously in VPENTA, we increase the size of array variables by adding prime number to avoid it intentionally[4].

A secondary cache miss is assumed to be delayed by as much as 30 cycles, when it tries to access memory, and a primary cache miss is assumed to spend 13 cycles. To handle prefetching, the hardware has a prefetch issue buffer, which can hold up to 16 prefetching requests.

Processor is assumed to have an on-chip primary data cache of 2K bytes, and a secondary cache of 64K bytes. Both caches are direct-mapped and use 8 bytes blocks. The size of array elements that is mentioned in this paper is assumed to be 4 bytes. The primary cache is operated by write-through manner, and the secondary cache by write-back manner.

4.2 Evaluation

In each bar of all graphs from Figure 7 to 10, the bottom section is the amount of time spent executing instructions, and the upper section is the processor idle time due to memory access. We use the following characters to denote the prefetching method applied to experimented loops: I, S, P. Each of these represents indiscriminate prefetching method, selective prefetching method and proposed one repectively. Character N denote a loop which is not reorganized.

Figure 7 presents the performances of LLL1, LLL 7, LLL 9, LLL 11 and LLL 18. Since LLL 18 is a nested loop with depth 2, we can see rather good performance by avoiding the insertion of unnecessary prefetching instructions for arrays with LCD.

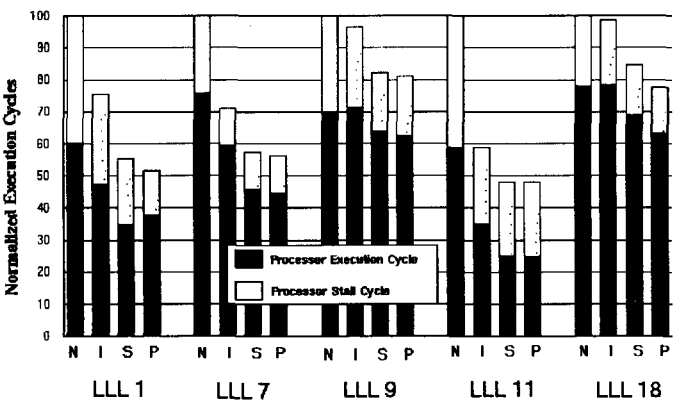

Fig. 7. Normalized execution cycles of simulated loops

Figure 8 shows the simulation results of LLL 21, the loop of matrix multiplication. Only three arrays are accessed within the body of LLL 21. We applied four prefetching methods to LLL 21 with different number of iterations. From the rightmost to the leftmost of each title, each number represents a loop bounds of the each loop nest from the innermost to the outermost. Cache can hold all arrays accessed within a loop body when the loop bounds of the innermost loop is 100. Meanwhile, as the iterations of the innermost loop is larger and larger, proposed method shows somewhat better performance than the selective method.

Figure 9 shows the simulation results of VPENTA. Since VPENTA is composed of only eight arrays with non-sequential memory reference pattern, heavy conflict comes to appear at run time. Henceforth, in Figure 9, we can find it that only a few cycles are saved with increasing of the iteration of innermost loop. Leftmost four graphs in Figure 9 is the results of simulations in case that array variables in VPENTA has 64×64 elements. Since such small number of array elements can be loaded in cache without conflict,

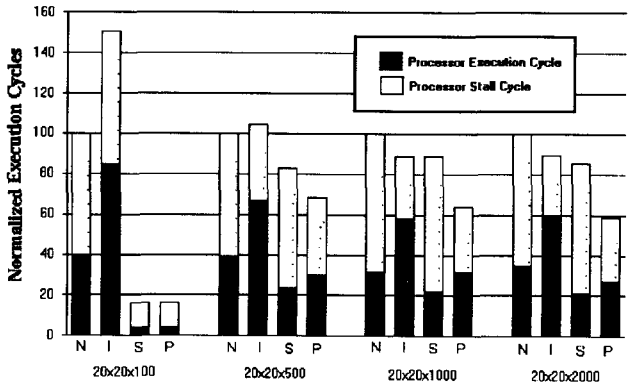

Fig. 8. Normalized execution cycles of LLL 21

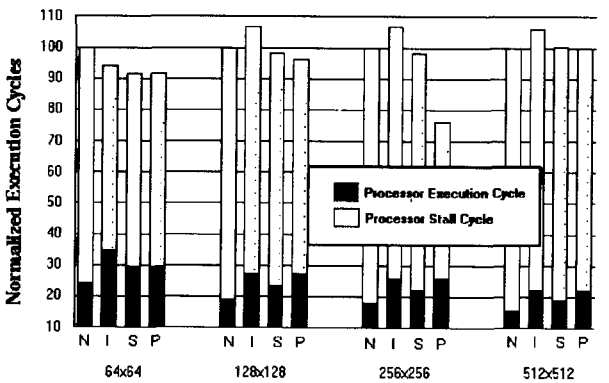

Fig. 9. Normalized execution cycles of VPENTA

Figure 10 shows a percentage of unnecessary prefetching instructions. Our proposed algorithm is more wasteful than the selective one. However, as the iteration of loop is larger and larger, the percentage of unnecessary prefetching instructions are getting smaller and smaller.

5 Conclusion

In this paper, we propose a software data prefetching mechanism to mitigate the problem which can occur by large array. To attack the problem of the failure of reusng data which is accessed non-sequentially, proposed algorithm has two opposite tactics: downright ignorance of it[1] and exhaustive consideration[4]. Additionally, our mechanism can successfully avoid inserting unnecessary prefetching instructions.

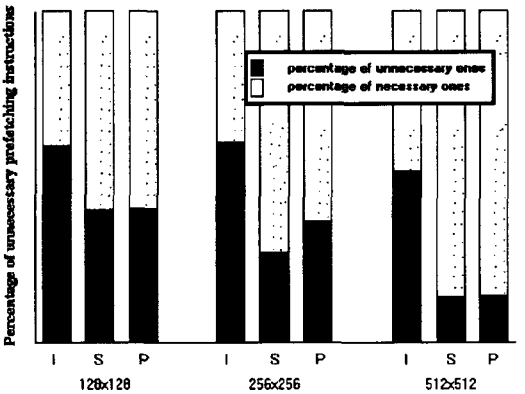

Fig. 10. Percentage of unnecessary prefetching instructions (VPENTA)

We realize our proposed algorithm into a preprocessor,*LOOP*[6]. After simulation using it, we can see that our method shows better performance than previous prefetching methods.

In the future, we will consider the type of the reference to an array, *load* or *store*. Relying on these types, the prefetched elements will be appropriately kept within a hardware which we plan to design and add. This extra hardware holds some prefetched array elements, and manages it in cooperation with a secondary cache. The preprocessor,*LOOP*, should transform a loop in the consideration of this hardware.

References

1. D. Callahan, K. Kennedy and A. Porterfield: Software prefetching. Proc. 5th ASPLOS. (1991) 40-52
2. Chen, T. F.: Data Prefetching for High-Performance Processors. TR 93-07-01, Dept. of Computer Science and Engineering, University of Washington, (1993)
3. Fu, J. W. C. and Patel, J. H.: Data Prefetching in Multiprocessor Vector Memories, Proc. 18th ISCA, (1991) 54-63
4. Mowry, T., Lam, M. S. and Gupta, A.: Design and evaluation of a compiler algorithm for prefetching. Proc. 5th ASPLOS. (1992) 62-73
5. M. E. Wolf and M. S. Lam: A Data Locality Optimizing Algorithm. Proc. SIGPLAN '91 PLDI (1991) 30-44
6. Se-Jin Hwang and Myong-Soon Park: Loop Reorganizing Algorithm for Data Prefetching. TR KUCS-CS-94-004, Korea University, (1994) (in Korean)