SOME ADDITIONS TO HARDWARE-BASED DATA PREFETCHING

Alaa R. Alameldeen and Layla Abou-Hadeed

Department of Computer Science and Automatic Control,
Faculty of Engineering, Alexandria University, Egypt

ABSTRACT

Data prefetching is used to increase the availability of data in the cache memory so that it is present when needed. Data prefetching is implemented using hardware or software techniques. Its basic idea depends on keeping track of data access patterns of a program and using this information to anticipate the location of data that is going to be accessed next. Chen et. al. introduced three variations for hardware-based data prefetching. Their designs handle the scalar, zero-stride and constant-stride access patterns. In this paper, a modification is added to account for the linear stride case. We are specifically interested for the linear rates of 1/2 and 2 which are the most frequent and do not involve great implementation overhead. This extension is better suited for application programs that exhibit a sufficiently large number of linear-stride data access patterns of the linear rates 1/2 and 2, such as index searching, to compensate for the increased hardware complexity and extra prefetched data. Preliminary experiments were held to explore the performance of the proposed design and produced satisfactory results.

Keywords: Prefetching, hardware-based prefetching, data cache, reference prediction.

INTRODUCTION

Due to the increasing speed gap between processors and memory, the increasing speed of the processor cannot be exploited efficiently unless some way of speeding up data access from the memory exists. The invention of cache memories as an intermediate storage between main memory and the processor has been a breakthrough, which helped to reduce the gap between the processor speed and that of main memory. However, memory latency cannot be totally eliminated. Upon a cache miss, the processor has to wait until the data is fetched both to the cache and to the processor.

Several methods have been developed to handle the problem of relatively high memory latency. Some of these methods are [1]:

- Data prefetching, that is fetching the data to the cache before it is actually needed by the processor. This can be done in hardware, i.e. hardware-based data prefetching [1, 2], or in software i.e. software-based data prefetching [2, 3 and 4].
- Cache hierarchies
- Lock-up free caches [5].

Hardware-based data prefetching is based on adding a small hardware unit that fetches data to the data cache before it is needed. Prefetching data to the cache depends on the previously accessed data patterns. Prefetching, however, cannot eliminate memory latency completely due to the unpredictable access patterns, as for example data-dependent addresses [1].
Software-based prefetching, is based on the analysis of the static program. An intelligent compiler may insert instructions that prefetch data many cycles in advance of their use by other instructions, so the program speed is not reduced [2, 3, 4]. These techniques can get more prefetches and can help in prefetching complex access

patterns (which cannot be done in hardware due to the high complexity), but the cost is the addition of prefetch instructions and calculations of the prefetched addresses, which may reduce the program speed.

Consider a program segment of nested loops, the memory access patterns can be divided into the following four main categories [1]:

1. Scalar, which is a simple variable reference, that does not change with respect to the loop index.
2. Zero stride, which is a reference inside an inner loop with a subscript expression that does not change with respect to the loop index (but may change with respect to the outer loop).
3. Constant stride, which is a reference inside a loop with a subscript expression that increases with a constant rate with respect to the loop index, such as the reference A[i] inside the loop of index i.
4. Irregular, which is any pattern other than the previous patterns.

Chen *et. al.* [1] introduced three variations for hardware-based data prefetching: Basic, lookahead and correlated methods. These variations handle the scalar, zero-stride and constant-stride access patterns. The basic scheme depends on the construction of a reference prediction table (RPT) for the instructions in the program that reference memory. An entry in the RPT consists of the instruction address, the previous address of the referenced data by this instruction, and the stride, which is the difference between the last two referenced addresses. In addition, an RPT entry contains a state field that provides information about the success of previous prefetches for this entry. Data prefetching is triggered when the program counter reaches an instruction that has a corresponding entry in the RPT. If the state of the corresponding entry indicates that prefetches can be predicted, the data at address (*previous address+stride*) is prefetched to cache.

This paper introduces a modification to the basic scheme so as to handle the linear-stride case. Specifically, the linear rates of 1/2 and 2 are considered. Preliminary results show the advantages of this modification for application program that exhibits a sufficiently large number of linear-stride data access pattern.

The rest of the paper is organized as follows: the next section introduces the linear stride memory access pattern. The section to follow presents the proposed modification to handle linear-stride prefetching. Then, we provide a comparison between the basic and the modified schemes. The last section presents the conclusions and future extensions.

LINEAR STRIDE: ANOTHER MEMORY ACCESS PATTERN

This pattern is a subset of the irregular pattern as classified before. In this access pattern, the memory is accessed, not by a constant stride, but by a stride that changes with a fixed constant. Specifically, when a constant s is multiplied by the stride each time, the next address referenced at this location is equal to:

previous address ± stride * s

and stride is updated each time by multiplying it with s. This means that we need to prefetch the contents of two addresses.

An example of this access pattern is in the famous binary search. In this program, the reference pattern for the middle of the current array is typically a linear stride for $s = 1/2$. An example for this reference pattern is (for an array of subscript ranging from 1 to 1000):

$500 \rightarrow 250 \rightarrow 125 \rightarrow 62 \rightarrow 31 \rightarrow 46 \rightarrow 54 \rightarrow 58 \rightarrow 56 \rightarrow 55$

A HARDWARE SCHEME FOR LINEAR-STRIDE PREFETCHING

The scheme provided here handles only the linear-stride case that varies by a

constant rate of 2 or 1/2. The reasons for this are:

1. Other constants require a multiplication operation to calculate the stride and the prefetched address each time. Multiplication is a slow operation that eliminates the performance gain due to prefetching. The multiplication by 2 or 1/2 can be implemented using a shift register.
2. The constants 2 and 1/2 are the most common constants in programs exhibiting linear strides (as in the binary search).

The new scheme (called the **modified scheme**) is a modification for the basic scheme of Chen *et. al.* [1]. We adopt the RPT proposed by them with some modifications. The following is a description of an entry in the RPT:

•**tag:** The instruction address.

•**prev_addr:** The last address that was referenced when the program counter reached that instruction.

•**stride:** The difference between the last two referenced addresses.

•**stimes:** The direction of shift (for stride) to implement multiplication by 2 or 1/2. It can take the following three values:

0 : no shift, where the stride is not multiplied by any constant (as in the previously discussed scalar, zero-stride and constant-stride patterns).

1 : shift left, where the stride is multiplied by 2.

-1 : shift right, where the stride is multiplied by 1/2.

stimes is updated by the absolute ratio between the old stride and the new stride. If this ratio = 2, stimes = 1. If this ratio = 1/2, stimes = -1. Otherwise, stimes = 0. The ratio calculation is implemented by two comparators and a decoder. The two comparators compare the new stride with the old stride shifted to left or right by 1 and the decoder is used to control the change in stimes.

•**state:** The past history of the memory reference pattern of this instruction. A state can be one of the following:

1. init: When a new entry is inserted in the RPT, or when a steady reference deviates from its steadiness.

2. transient1: When the stride of the (init) state is incorrect, so the system is not sure whether the memory access pattern of this instruction is a constant stride, linear stride or irregular.

3. transient2: When the stride of the (transient1) state is incorrect, so the system is not sure whether the memory access pattern of this instruction is a linear stride or irregular.

4. steady: The prediction is stable (stride is either 0 or constant or linear with stimes 1 or -1).

5. no_pred: No prediction is made because three successive guesses were incorrect.

A prefetch of data is triggered when an address is reached that has a corresponding entry in the RPT. First, prev_addr is set to the last referenced address. The addresses to be prefetched are then calculated as follows:

1. If stimes = 0, and state ≠ no_pred:
the address to be prefetched = *prev_addr + stride*

2. If stimes = 1 or -1 and state ≠ no_pred:
the addresses to be prefetched = *prev_addr ± (stride shifted in the direction of stimes)*
and the new stride is equal to stride * 2 (stride / 2) if stimes = 1 (-1).

3. If state = no_pred, no data is prefetched.

The state transition diagram of this scheme is shown in Figure 1. A simple hardware block diagram for the implementation of this scheme is shown in Figure 2.

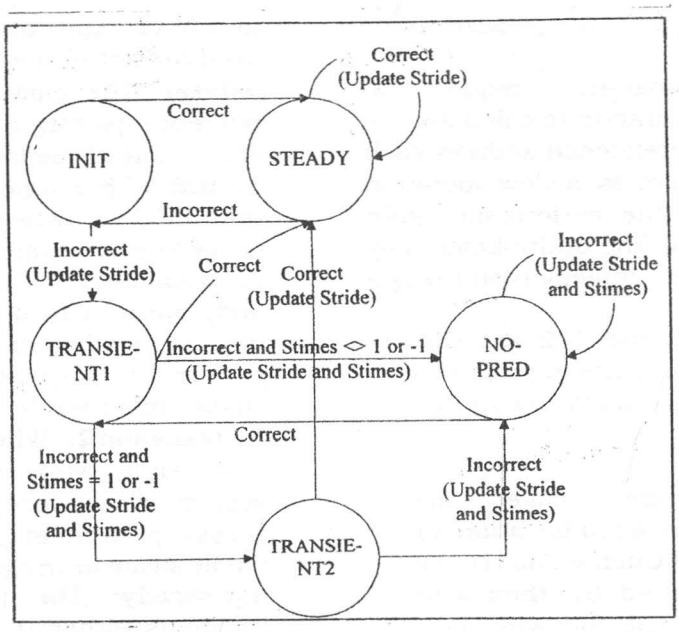

Figure 1 State transition diagram for the modified scheme

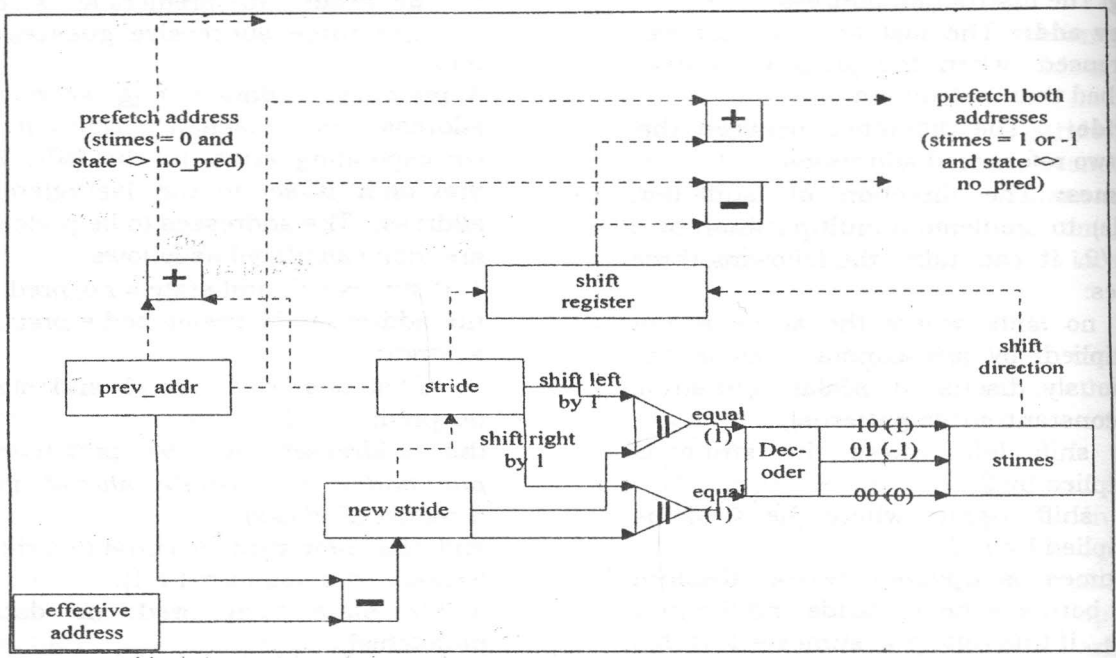

Figure 2 A block diagram for implementing linear-stride pre fetching. The solid lines are activated first then the dotted lines.

Example

Assume that address 100 in a certain program references data of the following addresses respectively: 256, 128, 192, 160, 144, 152, 156, 154, 155. Table 1 shows the values in the RPT entry of tag=100, and the prefetched address(es) (in the column *Prefetches*) corresponding to each referenced address (in the column *Reference*). It has to be noted that most referenced addresses are prefetched before being actually referenced.

Table 1 Contents of the RPT entry in the example

Reference	prev_addr	stride	Stimes	new state	prefetches
256	-	0	0	init	-
128	256	-128	0	transient1	0
192	128	64	-1	transient2	224, 160
160	192	-32	-1	steady	176, 144
144	160	-16	-1	steady	152, 136
152	144	8	-1	steady	156, 148
156	152	4	-1	steady	158, 154
154	156	-2	-1	steady	153, 155
155	154	1	-1	steady	-

COMPARISON BETWEEN THE BASIC AND MODIFIED SCHEMES

Extra hardware

The modified scheme requires the following additions to the basic scheme hardware. An extra bit for the state is needed, due to the added state. Extra two bits for the stimes field are also required in each RPT Entry. An adder and a subtractor are required to calculate the addresses to be prefetched. A shift register that can shift either to the left or to the right is also needed, as well as two comparators and a decoder. The connections to all stride and stimes fields are added as required.

Simulation of the Basic and Modified Schemes

Reference patterns were extracted from some of the most famous application programs that use array references. The simulation program reads these memory references and simulates the action of the basic and modified schemes.

The algorithm of simulation is as follows:

```
INPUT Parameters:
    - Memory Reference Array  ( R ). Each entry
          contains the program address (addr) and the
          referenced address (RefAddr)
    - number of references (nref)
    -method : 1 for Basic and 2 for modified scheme.
OUTPUT Parameters :
    - nfound : number of entries found in Cache
    - nnotfound: number of references that are not
          found in Cache
    - npref : number of Cache prefetches triggered.
PROCEDURE
1. nfound := 0;
2. nnotfound := 0;
3. npref := 0;
4. InitializeCache;
5. InitializeRPT;
6. For i := 1 to nref do
  a. RefAddr := R[i].RefAddr;
  b. if RefAddr = 0  {Program references ended}
      then begin
              InitializeCache;
        InitializeRPT;
           end
       else begin
            addr := R[i].addr;
            If AddrInCache (addr)
              then begin  {address is found in cache}
                    inc (nfound);
                    UpdateRPT (RefAddr, addr, npref,
                              true , method);
              end     {Address is in cache}
              else begin   {address is not found in cache}
                    inc (nnotfound);
                    FetchToCache (addr);
                    UpdateRPT (RefAddr, addr, npref,
                              false , method);
               end;    {Address is not in cache}
      end;
```

The function (AddrInCache) checks a reference address to determine whether it is in the cache or not. The procedure (FetchToCache) fetches data at a certain address from memory to cache. The function UpdateRPT operates exactly as the previous state transition diagram.

Reference patterns were extracted from the following programs.

1. Bubble sort (array size = 50)
2. Insertion sort (array size = 100)
3. Shell sort (array size = 100)
4. Quick sort (array size = 100)
5. Quick sort (array size = 200)
6. Merge sort (array size = 100)
7. Merge sort (array size = 200)
8. Heap sort (array size = 100)
9. Heap sort (array size = 200)
10. Matrix multiplication (size 50x50)

Assuming that the prefetched block size = 1 array element, the simulation results are shown in table 2.

Table 2 Simulation results of some programs

Program	Cache Hit Rate (%)		Number of Prefetches	
	Basic	Modified	Basic	Modified
1	99.96	99.96	49	49
2	99.96	99.96	98	98
3	99.84	99.84	117	117
4	99.82	99.82	103	104
5	99.92	99.92	205	204
6	97.72	97.76	151	153
7	98.05	98.15	306	315
8	99.89	99.89	99	103
9	99.95	99.95	198	201
10	96.00	96.00	127594	127594

It has to be noted that for the programs that have irregular memory access patterns (merge sort for example), the modified scheme provided slightly better results.

For the binary search program, which exhibits the most benefits, the results are summarized in Figures 3 and 4 and Table 3. Figure 3 shows obviously that the modified scheme provides very much better performance. Figure 4 shows that cache hit rate improves with the increase in the prefetched block size. This is because fetching a bigger block means more of the adjacent data being fetched and less prefetches required. Table 3 shows the ratio between the number of prefetches in the modified and basic schemes. It indicates that the modified scheme includes many more prefetches than the basic scheme. However, many of these additional prefetches are correct and this gives a performance gain.

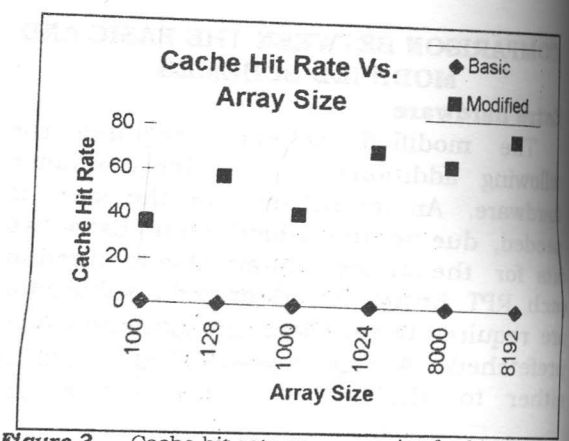

Figure 3 Cache hit rate vs. array size for binary search (prefetched block size =1 array element)

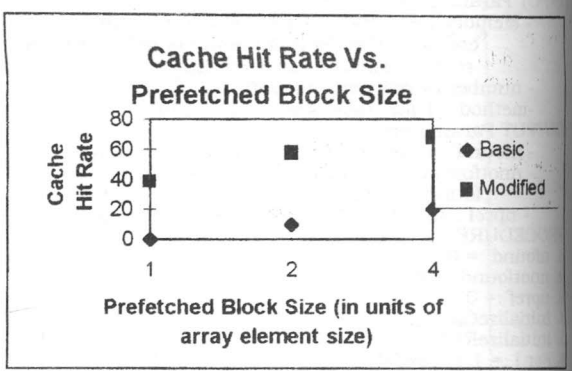

Figure 4 Cache hit rate Vs. prefetched block size for binary search (array size = 1000)

Table 3 Ratio between the number of prefetches in the basic and modified schemes for Binary Search

Program	Modified : Basic
Size =100, block=1	10.98
Size =100, block=2	9.18
Size =100, block=4	6.70
Size =128, block=1	14.30
Size =1000, block=1	12.85
Size =1000, block=2	23.21
Size =1000, block=4	21.44
Size =1024, block=1	26.00
Size =8000, block=1	41.69
Size =8000, block=2	37.88
Size =8000, block=4	37.70
Size =8192, block=1	38.00

CONCLUSIONS

Preliminary results show that for all the experimented programs, the modified scheme do not reduce the cache hit rate, and for most of them, it increases the number of cache prefetches, getting many additional correct prefetches. When the prefetched block size increases, the number of prefetches decreases and the cache hit rate increases.

In programs that exhibit a large number of linear strides of constants 2 and/or 1/2 (such as applications using the binary search), the modified scheme is much better than the basic one. In such programs, the proposed modification is cost-effective.

Future Extensions

This work needs to be extended by constructing a better simulation model for cycle-by-cycle operation of processors and performing a thorough analysis of the modified scheme based on some realistic benchmarks. The lookahead and correlated scheme and their corresponding modified schemes also need to be further studied.

REFERENCES

1. Tien-Fu Chen and Jean-Loup Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Transactions on Computers, Vol. 44, No. 5, pp. 609-623, (1995).
2. P. Vander Wiel Steven and David J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques", IEEE Computer, Vol. 30 pp. 23-30, July (1997).
3. William Y. Chen, Scott A. Mahlke, Pohua P. Chang and Wen-Mei W. Hwu, "Data Access Microarchitectures for Superscalar Processors With Compiler-Assisted Data Prefetching", Proc. 24th Annual International Symp. Microarchitecture, pp. 69-73, November (1991).
4. T. Mowry, M. S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", Proc. Fifth International Conf. Architectural Support for Programming Languages and Operating Systems, pp. 62-73, (1992).
5. Tien-Fu Chen and Jean-Loup Baer, "Reducing Memory Latency Via Non-Blocking and Prefetching Caches", Proc. Fifth International Conf. Architectural Support for Programming Languages and Operating Systems, pp. 51-61, (1992).

Received August 13. 1999
Accepted September 18. 1999

بعض الإضافات إلى طرق الجلب المسبق للبيانات المنفذة بمكونات مادية

علاء علم الدين و ليلى أبو حديد
قسم الآلات الحاسبه و التحكم ىلآلى - جامعة الاسكندرية

ملخص البحث

يستخدم الجلب المسبق للبيانات من أجل زيادة احتمال تواجد البيانات فى الذاكرة المختبئة وقت الاحتياج إليها، ويعتمد على متابعة أنماط أماكن البيانات التى يرجع إليها البرنامج واستخدام الأماكن السابقة للتنبؤ بالأماكن التى سيتم الرجوع إليها فيما بعد. وقد قدمت بعض الطرق التى يتم تنفيذها بدوائر داخل المعالج الدقيق من أجل الجلب المسبق للبيانات، حيث قدم تشين ثلاثة منها، وعالجت هذه الطرق ثلاثة من أنماط الرجوع لأماكن البيانات وهى : الرجوع إلى نفس المكان والرجوع لمكان بخطوة تساوى صفرا والرجوع إلى مكان بخطوة ثابتة، ويقدم هذا البحث تعديلا لمعالجة نمط الخطوة الخطية، حيث يرجع البرنامج إلى مكان بخطوة تتغير بمعدل ثابت. ويهتم البحث بالتغير بمعدل ٢ أو ١ \ ٢ وهى أكثر المعدلات شيوعا وأبسطها عند التنفيذ، ويفيد هذا التعديل البرامج التطبيقية التى تحتوى عددا كبيرا من المعدلات الخطية للتغير. وقد تم إجراء بعض التجارب المبدئية والتى حققت نتائج مرضية.