블록 구조형 멀티미디어 데이터의 선인출

정회원 김 석 주*, 준회원 이 병 권**, 정회원 김 석 일**

A Block Structured Multimedia Data Prefetching

Suk-Ju Kim* ***Regular Member*****, Byung-Kwon Lee**** ***Associative Member*****,**

Suk-Il Kim** ***Regular Member***

요 약

스트리밍 형태로 처리되는 멀티미디어 응용 데이터는 공간적 지역성은 큰 대신 시간적 지역성이 낮은 특징이 있다. 이 논문에서는 동적 선인출 기법을 작용하여 시간적 지역성은 낮으나 공간적 지역성이 있는 멀티미디어 응용 데이터에 내재된 메모리 참조의 규칙성을 활용하는 기법을 제안하였다. 특히 제안한 기법은 배열을 작은 블록으로 나누고 작은 블록별로 계산을 수행하는 응용 프로그램의 경우에 기존의 방법과 비교하여 선인출 에러를 크게 줄일 수 있는 특징이 있다. 여러 가지 미디어 벤치마크에 대한 실험 결과 제안한 기법이 기존의 선인출 기법에 비하여 선인출 에러 발생 회수가 크게 개선된 것을 확인할 수 있었다.

ABSTRACT

As to medium data which is involved in the form of streaming for a multimedia application, it characterizes that spatial locality occurs strongly but temporal locality appears even weaker. In this paper, with regard to dynamic prefetching, we suggest a method to make the most of memory reference regularities which typically innate by nature in the multimedia data with strong spatial locality but with weak temporal locality. Especially, the suggested method has a remarkable capability such that it can reduce prefeching errors substantially compared to existing prefetching methods for an application program which divides an array into small sub-blocks and, plus executes in the unit of sub-block. We carried out experiments to test the suggested method using various MediaBench benchmarks. From the results, we have confirmed that the occurrences of prefetching error decrease effectively than those of existing linear prefetching methods.

I. Introduction

Referring to Moore's law, processor speed doubles regularly in every 18 months. However, memory speed has been improved with the rate of only one tenth a year. From

* 혜천대학 컴퓨터・멀티미디어 전공(kimse@hcc.ac.kr) ** 충북대학교 전기전자및컴퓨터공학부
김석일(ksi@cbucc.chungbuk.ac.kr), 이병권(sonic747@just.chungbuk.ac.kr)

논문번호 : 030398-0915, 접수일자 : 2003년 9월 15일
※ 본 연구는 한국과학재단 목적기초연구(R05-2002-000-01470-0)지원으로 수행되었습니다.

this gap, fast memory reference time becomes more crucial for sustaining the throughput upturn in the modern processors[1]. According to Fritts[1], memory reference time will take up a 25~50% on all execution time.

More than ever, multimedia applications increase for many reasons, to name a few, image/video compression, WWW and etc. The data related to multimedia application often have the form of large scaled array or streaming patterns[2][3]. This means that the required amount of cache system is much steep for obtaining a satisfying performance in multimedia applications.

For a problem involving references to data, it shows that an amount of required data increases sharply at one point. It is unlikely that the data required this way would be found in the cache. Concerning cache management, even after an event of replacement, the replaced data block is improbable to be looked for again before its removal. This is conflicting with the fact that caches work ordinarily on the basis of the temporal locality of program behavior[4][5]. Deteriorating performance stems from that memory reference is done in streaming patterns.

To deal with this problem, we concentrate our efforts on exploiting more refined pattern of memory references. In general, there is a good chance that multimedia data has a commanding locality in it[4][5]. And, memory reference tends to show an exceptionally strong regularity on all streaming data[3][6]. The reason is that the majority of multimedia operation is particularly simple in practicing a routine over multimedia data. The structure of data with streaming patterns usually represent array format and iterative block structure is positively prevailing concerning the means of handling arrays. That means by all means memory references also rely on and reflect those simple block patterns.

Prefetching systems give option about how to introduce new necessary block to cache system[7][8]. In recent years, a dynamic prefetching unit has been used as well alongside with original cache controller[9][10][11]. Dynamic prefetching unit is known as an effective remedy for the problem that seems to be common in multimedia data access: unnecessary data takes up too much cache space accidental or incidental way[12]. Dynamic prefetching unit is literally intended to bring a designated section of memory into cache in advance in order to meet the possibility of cache miss going over previous several references.

Our approach is to identify and confirm a block structure and relate the block structure to deciding a feasible block to be used. We propose a dynamic prefetching method which features regulations for catching up a block pattern through memory access process.

In section 2, we give a little information about some data prefetching methods related to our research. Especially, we figure out reference prediction technique (RPT) devised by Chen[9].

In section 3, we present a block reference prediction technique(BRPT). The feature of BRPT makes a difference in detecting strides changing according to a block format. Combined with typical rule, because of a few expanded controls over stride-changes, BRPT puts out proper prefetching address in most cases.

In section 4, we present experiment results. Experiments on the incidence of cache miss and the memory cycle to handle the cache miss indicate that BRPT is better than RPT in most occasions.

In section 5, we reached conclusion that suggested BRPT might be an especially good source for better performance considering that memory referencing to a sub block form is very common to multimedia applications.

Ⅱ. Related Work

For dynamic prefetching, hardware observes memory reference instructions and generates prefetching instructions under the guidance of certain rule. In this section, we explain about the working of some prefetching methods available at present in industry.

Those are one block look-ahead prefeching (OBL)[13], multiple block look-ahead prefetching[14], and reference prediction technique[9].

One block look-ahead prefetching (OBL) [13] is to prefetch one neighbor block with the demanded block reference. Thus, OBL actually requests upto two blocks at the same time. This feature of OBL would improve the performance dramatically while cascade memory blocks are refernced conseqentially, since no cache miss occurs except beginning of the computation. However, the prefetching block is deterministic, the block can not be referenced if memory reference patterns are not sequential. Such blocks cause a cache pollution, which implies an unnecessary and harmful cache replacement.

The cache pollution problem can be solved equipping a buffer between the memory blocks and the data cache[14]. The buffer temporarily stores all the prefetched blocks, and only the actually referenced blocks are moved into the cache. Thus, no unneccessary blocks can exist in the cache. Furthermore, we can request multiple block prefetch without any cache pollution by increasing the buffer size. However, we have to encouter a buffer block replacement policy to provide high performance. And the effectiveness of the buffer is still limited to applications that refer the memory blocks in sequential pattern.

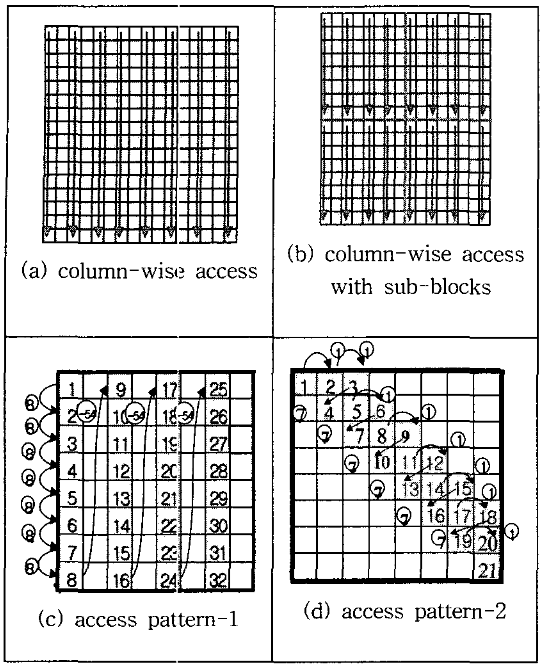
Reference prediction technique (RPT)[9] is to determined the prefetch block of a certain distance with a demanded block. The distance is calculated according to the previous references. Thus, this approach would provide superior prefetching hit for multimedia applications, since media data reference patterns in these applications are mostly linear. However, this approach still has weakness in expecting of prefetcing blocks whenever a big media data is partitioned into multiple sub-data and every sub-data is referenced by the application. More details will be discussed in the section 3. We propose a new methodology to expect prefetching blocks even sub-data referencing is used.

Ⅲ. Introducing New Techniques

3.1 Prospect of an additional regularity

RPT utilizes regulations with regard to the address of memory reference over the application program's run. Most applications can make a quality performance because they refer data elements with regularity. But, RPT has an inescapable shortcoming. It is that RPT interprets the regularity of memory address with the norm of 1 dimension. Because many applications involved in image handling or processing tend to engage the data represented with 2 or even higher dimension, the performance has a barrier with RPT. Fig 1 displays an example for the defect of RPT.

Such as JPEG encoding standard in which applying of DCT is supposed to be conducted by the unit of 8×8 pixel matrix for a given image, many multimedia applications will be based on those individual sub-blocks, across the data, rather than on the total acquired data during their executions. This is an eligible policy to guarantee the transferablility of the submission of one module's work to any other operational module in a multimedia application.



(a) column-wise access

(b) column-wise access with sub-blocks

(c) access pattern-1

(d) access pattern-2

Fig 1. Examples of memory reference

Fig 1(b) shows that data represented on a 2 dimensional array with the size of 16×16 is divided into sub-blocks with the size of 8×8.

Looking to Fig. 1 (a) and (b), this is a visual representation for a potential formation of memory references. Each cell represents a data element occupying a memory space. Without generality loss, we assume sequent memory address increases as row order in the memory space, and the succeeding memory address appears as adjoining cell in the right side in a row.

Comparing Fig 1 (a) and (b), you can see the shortcomings of RPT in case of sub-block in question. In Fig 1 (a) and (b), for each block, memory access is done successively in column by column order. Those access patterns may come up for a second matrix operand in matrix multiplication. The cell with pale color represents the situation in that prefetching address is proved not correct. Successful prefetching is represented as darker cell.

In case of access pattern in Fig 1(b), regularity of memory referencing degrades compared to (a). The sequence of reference memory address undergoes a breaking change by the very nature of array format as a new row or column corresponds to. From this particular instance, the accuracy of prefetching based on RPT decreases.

Because RPT decides a stride by looking at past trends, to mark a suitable stride, it must find at least two times of same distance in the successive memory references for a subjective memory reference instruction. This number of two occurrences must be a full complement of a stride properly marked. The value of the stride should stay right what it is, not changing it until finding out another value of stride. Even if a mismatch takes place one time, the marked stride should not change with the expectation to follow for the stride instead of the new value. This policy is advisable because you still have the prospect of informative stride unless you have failed twice.

This simple regulation makes prefetching easy. However, this is definitely is not sophisticated enough to sort out block patterns. More continual prefetching errors are occurring if sub-block pattern prevails.

In Fig 1(c) and (d), the number on a data element corresponds to the order by which the element have been chosen. In the RPT, out of the differences of the referenced memory addresses, it will find a stride and forecast on the next memory address and prefetch it as long as the stride remains unchanged.

Fig 1(c) numbers strides in the sequence: 8,8, ... ,(-54), 8, 8, ...,(-54). In this case, the next memory address can be forecasted as long as stride of 8 stays same. But, at the moment stride changes to-54, no forecasts will be possible because of the break of regularity. Fig 1(d) shows a different pattern, but with the same context as Fig 1(c), the strides appear to alternate in sequence.

Much of multimedia applications including audio/visual processing and MPEG encoder/decoder acquire some improvement by pulling in a wide variety of resources through various levels and measures. In this many-sided problem of enhancing, in terms of the significance of the structure, for an example, one method[15] observes incidence of area-linked metamorphosis along DCT process and detect the sign of sparse matrix format targeting to alleviate computational cost in the motion estimation step in MPEG encoding.

At work, such type of enhancement is prone to be tied to specific purpose or require a particular hardware which combines the unique characteristic with good results. With the help of our suggested method, there is possibly room for a particular method or hardware to get further improvement of performance because our suggested method aims anything different, that is to improve cache performance separately and apart from the special method.

3.2 Collecting another frequently-changing stride

In Fig 2, reference prediction table has 4 main components. The stride segment shows the distance of the data under consideration. The state field supports the acquisition of proper stride. Such reference prediction table has an additional field to mark a state in a conjectural procedure for making stride right on. The tag field corresponds to the program counter of the instruction which has been chosen. It is used for the distinction of each memory reference instruction. Over duration of repetitive references, a memory reference instruction can bring about a change in stride. In order to get a correct stride, a field for storing the previous referenced address for the corresponding instruction is required.

For existing RPT, the table has one stride field and one state field. For our suggested method, the corresponding field expands to have two fields. Fig 2 shows comparisons of table of RPT vs. table revised by our suggested method.

L1_stride field is updated in time of the repetition for a memory reference instruction. If the value of the field remains unchanged, L1_state field has the value of steady. If once unchanged L1_stride field changes, L1_state field switches to init which tells that the regulation becomes lost. At the same time, the value of L1_stride is to be copied to L2_stride field.

By the same way, whenever the value of L1_stride does not conform to the regularity established already, it is copied to L2_stride. By this means, the regularity such as 8,8,...,8 can be noticed by L1 fields and the regularity of -54 can be marked by L2 fields. Therefore, it may be considered that relevant regularities are detected throughout the whole memory reference.

tag	previous_address	stride	state

(a) Reference Prediction Table by Chen[9]

tag	previous_address	L_1		L_2	
		stride	state	stride	state

3.3 Reckoning of prefetch address

Moon[16] talked about the background information that will be effective to check out 2 or even higher dimensional block pattern. We find it easy that a sequence of successful prefetching by a regular stride conveys a message concerning the number of elements on a possibile row or column in block format.

To recognize a block format, BRPT collates the number of times in that a certain stride keeps up and steady hit of prefetching address continues. Then, it automatically produces a correct prefetching address discerning a new round according to the block format.

The practicability of BRPT can be stated; it forecasts the variations of strides in a successive memory reference separately-yet without any association or link to controlling structure such as history of branch instruction.

Fig. 3 shows an example for how BRPT proceeds. (Without loss of generality, we assume the stream of data is reference by row order.)

L2_state of first-change mode means that you are in the second row. So, when L1_state changes from initial to steady (You will have this change of mode if the next element keeps the L1_stride.), we have the clue to know of the number of elements in a row. Everytime as new memory reference is made, if L1_stride keeps unchanged and therefore L1_state is maintained as steady, and temporary variable hit_count is increased by 1.

The variable hit_count eventually records information related to number of element of the row. If L1_state becomes to be initial, this indication is interpreted as the beginning of third row. Therefore, still unchanged previous_address belongs to the last element of the second row and this value is stored in a variable temp_base, for the purpose of some checking for the 2 dimensional array structure which will be mentioned soon.

At this moment, L2_state of first-change makes to transient. Also, the difference between current referenced address and previous_address (presumed as the last element of the second row) is kept into L2_stride field. And also, update to previous_address is done and the next reference will be made.

In the situation with transient L2_state, every time L1_state changes to steady and stays as steady state, variable hit_count is decreased by 1. Therefore, in time hit_count becomes to 0, this sign indicates current referenced data takes the position corresponding to the last element of the third row.

In this moment, we go through a test for recognizing the structure of 2 dimensional array format. This test is done by comparing the difference between currently referenced address and temp_base to the difference between temp_base and address_keep.

If both of two values coincides, same distance at the breaking point (the occasion in which steady L1_state is replaced with initial) can be looked as that row-changing stride supposedly repeated twice and the number of element in this turn (supposedly row) is reached to the number of previous turn(row), although it is not certain that there still remains more elements suitable with L1_stride.

At this moment, the best thing you can do is to conjecture that you have found the regulation of 2 dimensional array structures. Here, you regard the current address as that of the last element of the third row. L2_state replaces transient by steady. Applying the value of L2_stride recorded previously, prefetching instruction becomes to be issued. But you do not know if the prefetching address using L2_stride instead of L1_stride is appropriate.

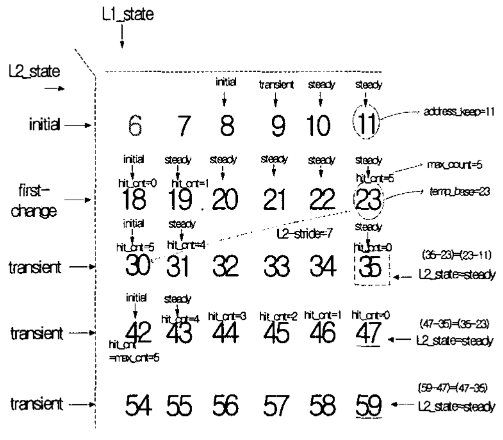


Fig 3. Example of memory reference

```
Set_prefetch_address() {
    if (flag for applying L2_stride is set)
        prefetch_address=address+L2_stride;
                otherwise  prefetch_address=address+L1_stride;
}
adjust_temporary_value() {
    if(L1_state is INIT)and(L2_state is FIRST-CHANGE)
        address_keep= previous_address;
    if(L1_state  is INIT)and (L2_state  is TRANSIT)
        temp_base = previous_address;
        max_hit_count=hit_count;
    if(L1_state is STEADY)and (L2_state is FIRST_CHANGE))
        hit_count++;
    if(L1_state  is STEADY) and (L2_state is TRANSIT)
        if((--hit_count) is 0) and
          (address is (2* temp_base — address_keep))
            set flag for applying L2_stride;
            L2_state=STEADY;
            address_keep=temp_base_address ;
            temp_base=address;
}
update_state() {
   if(L1_state  is STEADY) and (L2_state is STEADY)
      L1_state= INIT;  L2_state =TRANSIT;
   update L1_state according to the rule shown in transition
            diagram.
  As this update proceeds, if there is a change STEADY to INIT
    if((L1_state is INIT)  L1_state= FIRST-CHANGE;
    if((L1_state is FIRST-CHANGE)  L1_state= TRANSIT;
 }
```

```
Update_stride() {
      if(L1_state  is INIT) and (L2_state is TRANSIT)
          L2_stride=address-previous_address;
      In case prefetching was correct,
      if((L1_state is TRANSIT))
          L1_stride= address-previous_address;
      In case prefetching was incorrect,
      if((L1_state is TRANSIT) or (L1_state is NO_PRED))
          L1_stride= address-previous_address;
}
Block Reference Prediction {
     For a coming memory reference instruction,
         conduct its operation.
     Using PC address of the instruction as tag,
         Seek out relevant slot in RPT.
     If there is no corresponding slot,
         Reserve one slot (either replace one or
                          assign a vacant slot)
     Comparing prefetching address to current address of
         referenced element,
         check out whether prefetching is correct or not.
     Update L1_state and L2_state( call update_state()).
     Update L1_stride and L2_stride( call update_stride()).
     Set temporary variables necessary to searching for 2
         dimensional array structure
           ( call adjust_temporary_value()).
     Set prefetching address( call set_prefetch_address()).
     Issue prefetching instruction if necessary.
     Replace previous_address field with current referenced
         address.
}
```

Fig 4. Block Reference Prediction Algorithm

You will get the actual memory address as you move on to next element. If this address is matched to the prefetch address, this implies that the currently referenced address can be regarded as that of the first element of the fourth row.

You can keep up going after further continuous data streams and see if this may be a set up of 2 dimensional array format. Thus, the value of temp_base moves into address_keep and the value stored in previous_address goes to temp_base.

Also, value stored in max_count replaces hit_count. The steady of L2_state is forced to switch to transient state. Those moves are intended to prepare for sorting out forthcoming references.

The same procedure for identifying another row continues as described above.

During the whole procedure above, at the occasions which do not keep a promising sign for finding a 2 dimensional structure, for one, bothering in the counting down on hit_count to zero or stray from the conditions with temp_base and address_keep, transient L2_state turns into initial. This means that it is going to scout to pick out another possible 2 dimensional array structure and goes on with above procedure again.

Fig 4 shows an outlined algorithm of Block Reference Prediction. It shows the working of BRPT unit including how to observe the sequence of referenced address and get L1_stride and L2_stride and apply each feasible and corresponding stride depending on the location in a block format if

	description	program module	data source	input size	image/frame size
MPEG	transforming tool for digital video stream	mpeg2enc	image frames of Y.U.V format	4 frames	256×242
		mpeg2dec	mpeg video streams		
JPEG	compression standard for image	cjpeg	image files of .ppm format	100Kbytes	172×189
		djpeg	image files of .jpg format		
EPIC	image compressing tool	epic	image files of .raw format	60Kbytes	256×256
		unepic	image files of .E format		

Table 1 Benchmark programs

such format were recognized.

The state of BRPT unit changes according to the rule shown in update_state() routine in Fig.4. In the routine, as L1_state changes from steady to init (at the time L1_stride does not prove it correct), L2_state becomes assigned different value over the procedure to get L2_stride as shown in Fig. 3.

In this process, the unit keeps temporary values such as hit_count, temp_base address, max_hit_count and address_keep as shown at the routine adjust_temporary_value() of Fig. 4. Those values are required to find block pattern.

The value of (address-temp_base) holds the current distance between the elements at that the L1_stride becomes disrupted If this value is equal to such previous distance : (temp_base-address_keep), we can know that the distance repeats and a block pattern appears supposedly. The value of hit_count and max_hit_count relates to how many times L1_stride goes undisturbed. As the unit gets a right L2_stride, it can also decide the time to apply whether L1_stride or L2_stride using those values.

Ⅳ. Experiments

To analyze the performance of BRPT, a trace-driven simulation was carried out on Digital Alpha DEC machine to model cache. The initial phase of the experiment is to generate the instruction trace of an application program. Using this trace as input, it moves forward to simulate the cache operation.

For obtaining the trace, we used the ATOM simulator[17] for analyzing benchmark programs using object codes of Alpha machine. The simulation part for cache operation was implemented based on Dinero III[18] which is a trace-driven simulator developed by Univ. of Wisconsin.

The simulator was designed to receive information about the trace of memory reference instructions and gather data for the memory reference frequency for each load/store instruction, cache miss rate and other measures of cache.

To compare BRPT with other prefetching methods mentioned earlier, we made transformation to the cache simulator to accommodate different methods. General scheme in the chart represents that there is no use of any prefetching. For the benchmark, we selected MPEG, JPEG and EPIC from Media Bench[19] benchmark. Table 1 shows detailed specifications of benchmarks.

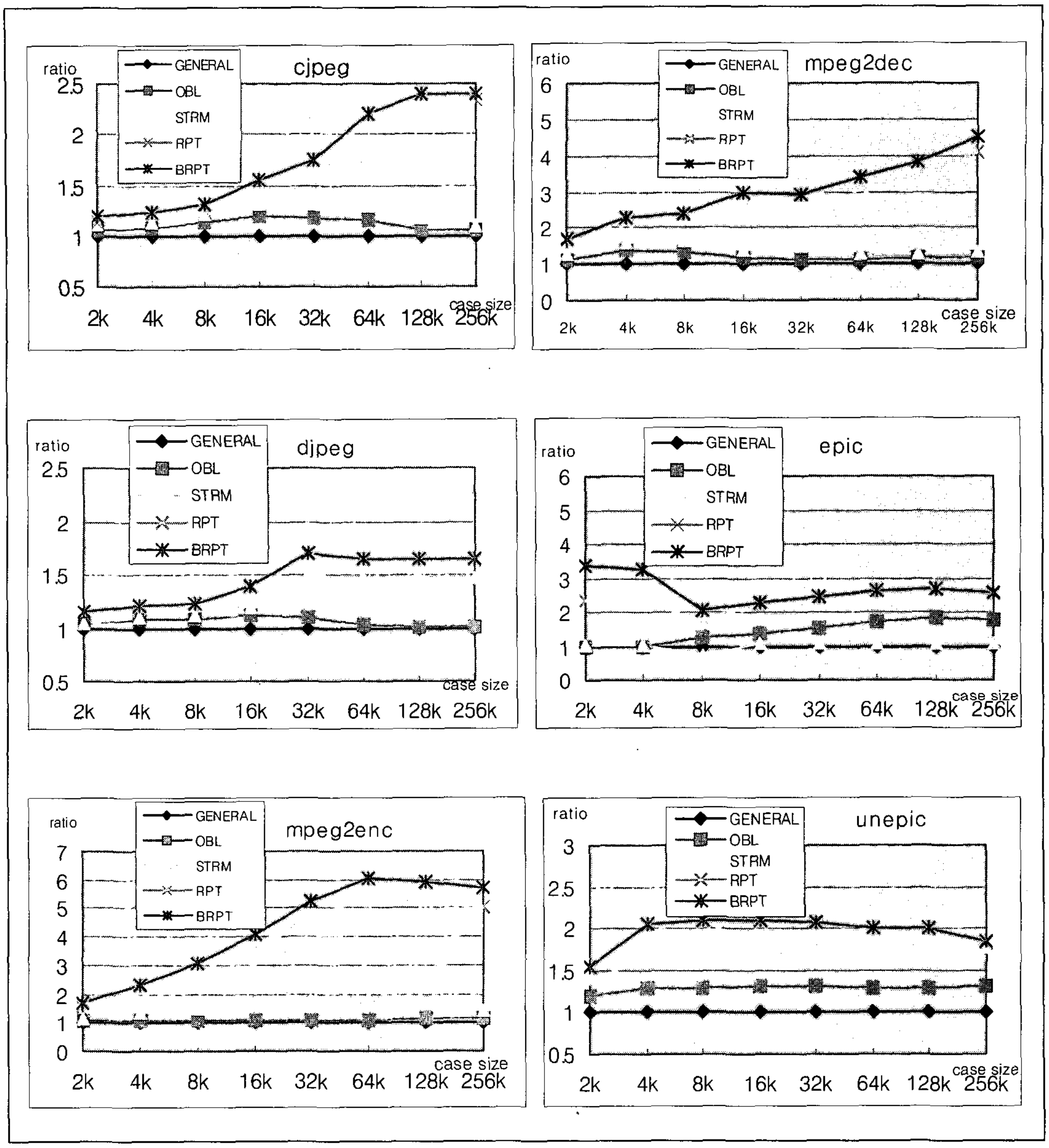


Fig 5. Cache miss improvement ratio

We made the evaluation in terms of two indices: cache miss rate and the cycle time for deferring to settle cache miss. Fig. 5 displays improvement ratio of cache miss frequency. This value has how much the corresponding prefethcing scheme might be better in terms of cache miss frequency than basic scheme that represents no use of prefetching.

In the below equation (1), *C.M.I.R.* shows the cache miss improvement ratio of the scheme.

$$C.M.I.R. = \frac{C_{BASIC}}{C_{THE\ SCHEME}} \qquad (1)$$

where C represents the cache miss frequency.

In terms of total memory cycles required to settle cache miss, we investigated with a similar method. The results of another index, the sum of deferred memory cycle time, showed up as almost same form as the charts in Fig 5. However, the improvement ratio of differed cycle appears with a slightly bigger percentage than that of the cache miss improvement ratio for each case.

We compare BRPT to RPT, using two indices. Fig. 6 shows cache miss- reduction ratio : R_c and memory cycle-reduction ratio : R_d .

$$R_c = \frac{C_{RPT} - C_{BRPT}}{C_{RPT}}$$

where C is the cache miss frequency.

$$R_d = \frac{D_{RPT} - D_{BRPT}}{D_{RPT}}$$

where D is the deferred memory cycle time to settle cache miss.

In case of mpeg2enc, BRPT has the most favorable outcome. It looks like that abundant regulations in the data before compression procedure make for the good result.

Comparing BRPT and RPT, throughout all benchmarks we have dealt with so far, R_c points to an average reduction of 4% in terms of cache miss frequency. BRPT outperforms RPT by 4%.

R_d measures average 5% having a positive effect on reducing memory cycles.

The reason for why the effect of BRPT measures differently in two indices may be related to exact timing at which prefetching instruction is issued and processed, that is, to bring memory block to cache actually. We will investigate further on this. From above indices, BRPT based on regulation seeking out 2 dimensional array form is expected to be better than RPT in most multimedia applications. Specially, for applications such as mpeg2enc and mpeg2dec, it makes for attractive option.

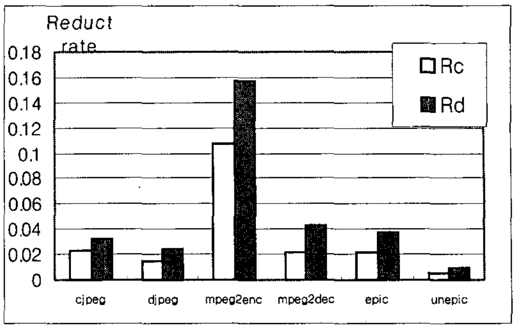


Fig 6. Relative qualities of BRPT

V. Conclusions

BRPT exploits a recognizable block pattern, which appears more in multimedia applications. BRPT and RPT have much in common; both have notion of predicting pefetching address based on stride. To check out the feasibility of BRPT, we examined on cache miss rates and memory cycles using benchmark programs in Media Bench benchmark. BRPT outdoes RPT in most cases. By logical reasoning, BRPT will become more effective of the two in the applications with more abundant occasions of block patterns.

Reflecting fact that a size of application enlarges and array size increases so quickly, BRPT is estimated to be beneficial and effective to make memory access time declined in multimedia applications.

REFERENCES

〔1〕 J. Fritts, "Multi-Level Memory Prefetching for Media and Streaming Processing," *Proceedings of International Conference on Multimedia and Expo*, pp. 101-104, August, 2002

[2] S. Carr, K. S. Mckinley and C. W. Tseng, "Compiler Optimization for

Improving Data Locality," *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 252-262, October, 1994

[3] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp.30-44, June, 1991

[4] J. R. Goodman, *Cache and Sequential Consistency*, Technical Report TR-1006, University of Wisconsin-Madison, February, 1991

[5] F. Harmsze, A. Timmer and J. Meerbergen, "Memory Arbitration and Cache Management in Stream-Based Systems," Proceedings of the DATE 2000, pp. 257-262, March, 2000

[6] C. K. Luk, Optimizing the Cache Performance of Non_Numeric Applications, Ph.D. Thesis, University of Toronto, 2000

[7] S. P. VanderWiel and D.J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *IEEE Computer*, Vol. 30, No. 7, pp.23-30, July, 1997

[8] K. Diefendorff and P. K. Dubey,"How Multimedia Workloads Will Change Processor Design," *IEEE Computer*, Vol. 30, No. 9, pp.43-45, September, 1997

[9] T. F. Chen and J. L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, Vol. 44, No. 5, pp. 609-623, May, 1995

[10] T. Horel and G. Lauterbach, "UltraSPAC-III: Designing Third-Generation 64-bit Performance," *IEEE Micro*, Vol. 19, No. 3, pp. 73-85, May, 1999

[11] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kupanek, F. X. Schumacher and J. Zheng, "Design of the HP PA 7200 CPU," *Hewlett-Packard Journal*, Vol. 47, No. 1, pp. 25-33, February, 1996

[12] D. F. Zucker, M. J. Flynn and R. B. Lee, "A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks," *Proceedings of International Conference on Multimedia Computing and Systems*, pp. 236-244, June, 1996

[13] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol 11, No. 12, pp. 7-21, December, 1978

[14] N. P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, May, 1990

[15] E. Feig and E. Linzer, "Discrete Cosine Transform Algorithms for Image Data Compression", *Proceedings Electronic Imaging '90*, pp.84-87, Novemver, 1990

[16] H. J. Moon, A Cache Managing Strategy for Fast Media Data Access, Ph. D Thesis, Computer Science Department, Chungbuk National University, March, 2003

[17] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the ACM SIGPLAN 94*, pp. 196-205, June, 1994

[18] M. D. Hill, *Dinero III Cache Simulator*, Technical Report, Computer Sciences Department, University of Wisconsin, Madison, 1997

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems," *Proceedings of the 30th Annual international Symposium on Microarchitecture*, pp. 330-335, December, 1997

김 석 주(Suk-Ju Kim) 정회원
1982년 2월 : 서울대학교 전자계산기 공학과 졸업
1984년 2월 : 한국과학기술원 전산학과(공학 석사)
현재 : 충북대학교 전자계산학과 박사 과정
1997년 9월~현재: 혜천대학 조교수

〈주관심분야〉 병렬처리

이 병 권(Byung-Kwon Lee) 준회원
1999년 2월 : 한밭대학교 전자계산학과 졸업
2002년 2월 : 한남대학교 컴퓨터공학과(공학 석사)
현재 : 충북대학교 전자계산학과 박사 과정

〈주관심분야〉 퍼지이론, 컴퓨터구조, 병렬처리, 임베디드 시스템

김 석 일(Sukil Kim) 정회원
1975년 : 서울대학교 전기공학과 (공학사)
1975~1990년 : 국방과학연구소 근무
1989년 : 미국 North Carolina 주립대학 (공학박사)
1990년~현재 : 충북대학교 컴퓨터과학과, 전기전자및컴퓨터공학부 교수
현재: 충북대학교 전기전자및컴퓨터공학부 학부장, 충북대학교 BK사업단장

〈주관심분야〉 병렬컴퓨터 구조, 슈퍼컴퓨팅, 병렬처리 언어, 시각장애인 사용자 인터페이스 등