Memory Access Pattern Analysis and Stream Cache Design for Multimedia Applications

Junghee Lee
The school of Electric Engineering and Computer Science
Seoul National University
Seoul 151-742, KOREA
konnen@iris.snu.ac.kr

Chanik Park
Software Center
Samsung Electronics
599, Shinsa-dong, Kangnam-ku
Seoul 135-893, KOREA
ci.park@samsung.com

Soonhoi Ha
The school of Electric Engineering and Computer Science
Seoul National University
Seoul 151-742, KOREA
sha@iris.snu.ac.kr

Memory system is a major performance and power bottleneck in embedded systems especially for multimedia applications. Most multimedia applications access stream type of data structures with regular access patterns. It is observed that conventional caches behave poorly for stream-type data structure. Therefore, prediction-based prefetching techniques have been extensively researched to exploit the regular access patterns. Prefetching, however, may pollute the cache if the prediction is not accurate and needs extra hardware prediction logic. To overcome these problems, we propose a novel hardware prefetching technique that is assisted by static analysis of data access pattern with stream caches. With the proposed stream cache architecture, we could achieve significant performance improvement compared with the conventional cache architecture.

I. Introduction

Memory system is a major performance and power bottleneck [9] in embedded systems as the performance gap between processors and memories is ever increasing. Though general cache schemes such as direct and set-associative caches can attenuate the gap on average, there is still much room for optimization if the memory access pattern is known a priori. In particular, multimedia applications (e.g. MPEG encoding/decoding, speech processing etc.) are characterized by their excessive data memory accesses but with regular access patterns for stream data structures. Therefore, we are concerned about memory system design for multimedia embedded systems.

Several researches have focused on prefetching techniques to exploit the stream-like access pattern. Prefetching can be triggered either by a hardware mechanism, or by a software instruction, or by a combination of both. The hardware approach detects accesses with regular patterns and speculatively issues prefetches at run time, which may cause some run-time overhead and a cache pollution problem in case of misprediction. On the other hand, the software approach relies on the compiler to analyze the program and to insert prefetching instructions, which may increase the code size.

In this paper, we propose a novel hardware prefetching technique that is assisted by a static analysis of data access pattern. We first obtain the physical memory trace of target multimedia applications and identify the data structures with regular access patterns in the source code level by static analysis. Then, we allocate each data structure into a special cache, called *stream* cache. A stream cache is similar to a conventional hardware prefetch buffer that fetches the data earlier than needed to reduce the cache miss penalty, based on the prediction of the next access.

It is, however, distinguished from the conventional prefetch buffer in two aspects. First, prediction is performed by the static analysis to avoid costly hardware prediction logic but with higher prediction accuracy. Second, each stream cache is assigned a different address space so that there is no cache pollution problem. Since a stream cache is an add-on feature to any existing architecture, the proposed technique is complementary to any previous efforts for optimal memory system design. More detailed comparison with related works is discussed in the next section.

Static analysis of data access patterns from the given memory trace and automatic synthesis of on-chip stream caches are two main themes of this paper. We identify three different types of access patterns and as many stream cache configurations for a set of popular multimedia applications. With the proposed stream cache architecture, we could obtain significant performance improvement compared with traditional cache architecture without any prediction logic.

The related works in the embedded memory system design are compared with the proposed technique in the next section. In section 3 we present the stream cache architecture template that our approach is based on. In section 4, our proposed methodology is explained. We present experimental results with a set of real multimedia applications, showing significant performance improvement over the existing cache configurations in section 5. Finally, we envision some perspectives of our approach and draw conclusions in section 6.

II. Related Works

While it has been a major topic for general purpose computer architecture with uncountable number of research results, optimal memory system design gains attention recently for embedded systems as a part of design space exploration. Unlike the general purpose systems, embedded systems have more chances of memory optimization because we may concentrate on the memory access patterns only for a given set of applications.

APEX approach is similar to ours in that it uses special

This work was supported by National Research Laboratory Program (number M1-0104-00-0015) and Brain Korea 21 Project.
The RIACT at Seoul National University provided research facilities for this study.

memory modules for predefined memory access patterns [5]. They have a library of special purpose memory modules in addition to a conventional cache; indirect reference module for linked list access pattern is an example. And, they categorize three kinds of memory access patterns: access patterns can be determined at compile time, access patterns the programmer has prior knowledge on, and access patterns that are complex and difficult to predict. Since they focus on the methodology, but not on memory modules themselves nor on access pattern analysis, our work is distinguished from their work.

Some hardware architectures to exploit the regularity of memory accesses have also been proposed for high performance computing [2][3]. Those proposed hardware architectures prefetch data with rather complex prediction logic, among which the preloading scheme is an important technique that many papers cited [3][11]. In this paper, we compare the performance of our approach with that of the preloading scheme.

The preloading scheme uses a reference prediction table to predict the next access. Each entry of the reference prediction table stores the PC of the load instruction, previous effective address, stride, and the state. If there is an entry of which PC is the same as the look-ahead PC, it prefetches the cache line. Look-ahead PC is the next PC expected to be executed. After every memory access it updates the reference prediction table. This scheme can exploit some of the regularity of the memory access. But it requires complex hardware logics and pollutes the cache if the prediction fails. On the other hand, our approach needs much less additional hardware without cache pollution problem.

There are also software techniques to prefetch data and instruction [4]. Those prefetch approaches are primarily based on prediction from the static analysis of the source code. In our approach, however, we prefetch data based on the static analysis of memory traces obtained from the dynamic execution profile through simulation.

Memory access pattern analysis has been also used for source code transformation or data layout optimization [10] [6]. Catthoor et al. observed that the source code should be first optimized to generate the optimal memory access patterns for a given memory architecture. They proposed several optimization techniques. For example, they pack the data structures according to their size and bit-width and allocate them into memory modules to minimize the memory cost [10]. They examine the memory access patterns in the source code to estimate the memory system performance and apply transformation techniques. Even though we also examine the memory access patterns for the data structures in the source code, we use physical memory address traces to obtain the prediction information for data prefetching.

Data layout optimization for memory hierarchy is a technique to reduce cache misses [6]. While previous researches mainly aim to reduce the conflict misses, our approach reduces mainly the capacity and cold misses [7] for stream-type data structures.

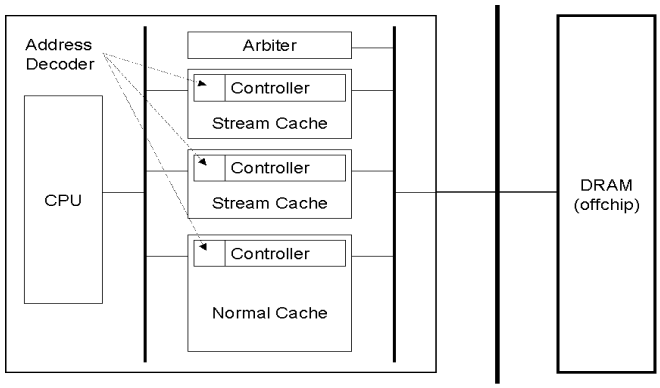


Fig.1. Architecture template

III. Stream Cache Architecture

A. *Architecture Template*

Fig. 1 shows the proposed memory architecture template that consists of a processor, a conventional on-chip L1 cache and a set of customized stream caches. Stream caches are assigned different address spaces from the conventional cache. A memory access request from the processor is routed to one of the caches based on the mapped address.

If a memory access occurs to a stream cache, the stream cache controller issues a prefetch request of the next address that is the sum of the current address plus the stride value stored in the *Stride* register inside the stream cache controller. There are other additional hardware logics; address decoders and bus arbiters for multiple caches. Even though they may increase the clock cycle, their effect is negligible in most cases [5]. Another possible side-effect of prefetching logic is the increased miss penalty of the conventional cache in case memory request is interfered by the outstanding prefetching request issued earlier.

B. *Stream Cache*

A stream cache module is parameterized by the number of banks, the line size, and the number of lines, which are determined by the memory access pattern analysis. A micro-architecture template of a stream cache module is displayed in Fig. 2. The line size is determined by the stride value. If the stride value is one, we can increase the line size to reduce the communication overhead. Otherwise, the line size is set to one to remove unnecessary prefetch overhead.

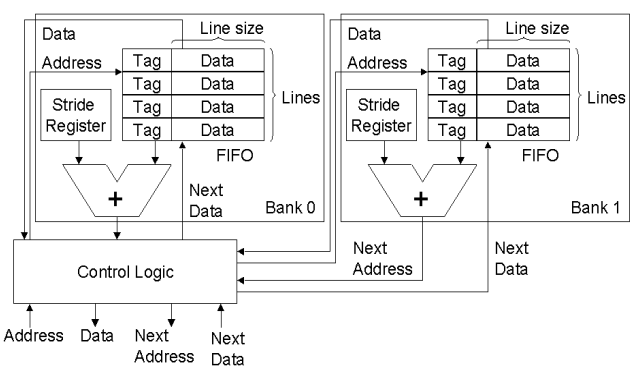


Fig.2. Stream cache micro-architecture

The number of lines is determined either by how many data should be prefetched or by how large working set should be preserved. In this paper, we assume that the number of lines is two for simplicity: one for current access and the other for the next access.

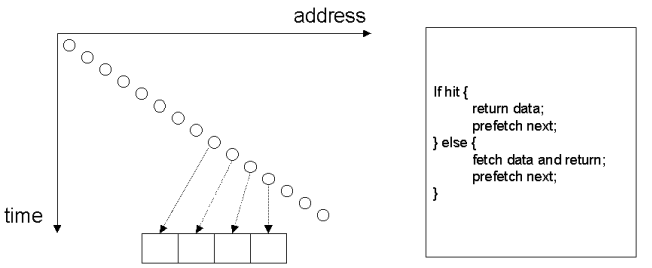
It is observed that three types of memory access pattern are frequent in multimedia applications. The stream cache modules are configured differently for each type. If the static analysis reveals another memory access pattern, a different stream cache configuration may be added. In this paper, however, we are concerned about three memory access patterns and the associated three types of stream cache configuration as shown in Fig. 3: fixed-stride, 2-way, and 2D stream caches. Fig. 3 also shows the pseudo code of the corresponding control logic of the stream cache.

Fixed-stride stream cache has the basic configuration that consists of a single bank and the fixed stride value is stored in the *Stride* register (Fig. 3(a)). If the current access fails, miss penalty should be paid to fetch the request data. Otherwise, the cache returns the requested data immediately from the cache, In both cases, it issues the next prefetch request whose address is the sum of the current address plus the stride value. This type of access pattern is typically observed at a one-dimensional bit stream buffer.

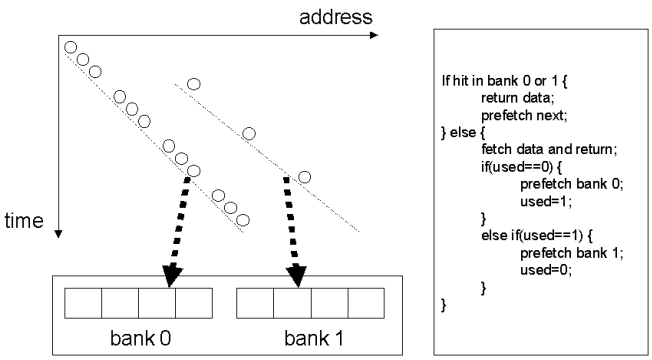
2-way stream cache has two banks of fixed-stride stream cache. Two banks have their own stride values while memory accesses to these banks are interleaved as illustrated in Fig. 3(b). We preserve two outstanding access locations for the same data structure. The flag *used* in the pseudo code of Fig. 3(b) indicates which bank is used last. This type of access pattern is typically observed at a buffer accessed in a nested loop structure.

2D stream cache has also two banks of fixed-stride stream cache. In this configuration, two banks are assigned two different sets of access regions in an alternating fashion as shown in Fig. 3 (c). After the last access of a region, the bank starts prefetching the next data of another assigned region while the current memory access is routed to the other bank. Thus, another *Region-Stride* register is needed to keep the distance between the assigned regions. The flag *used* in the pseudo code of Fig. 3(c) also indicates which bank is used last. This type of access pattern is typically observed at a 2-dimensional array.

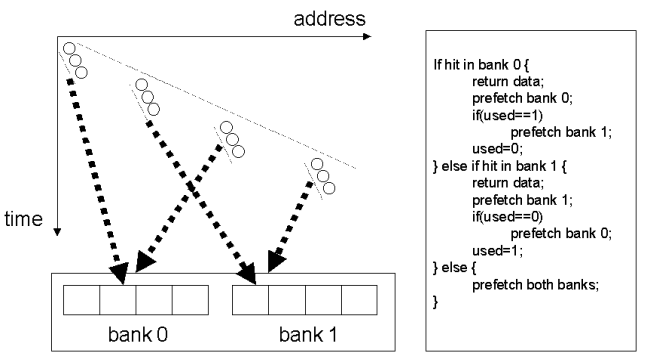
Fig. 4 presents an example code. There are three stream type data structures: *bitStream*, *qOutput*, and *mBlock*. The *bitStream* buffer is suitable for the fixed-stride stream cache and its stride is one. The *qOutput* buffer has outstanding two interleaved access locations between the inner and the outer loop body, so becomes the target of an 2-way stream cache. Buffer *mBlock* is a two dimensional array, thus suitable for the 2D stream cache. The inner loop defines a region of fixed-stride access while the outer loop guides the access region change. Since the distance between the regions is fixed, the value is stored in the *Region-Stride* register. If the memory access pattern to the *ctab* data structure is not identified as a known pattern, *ctab* is assigned to a normal cache. The resultant stream cache architecture is displayed in Fig. 5.



(a) Fixed-stride stream cache



(b) 2-way stream cache



(c) 2D stream cache

Fig. 3. Three kinds of access patterns and their mapping to the corresponding stream caches. The pseudo code of the associated controller logic of Fig. 2 is also shown.

```
for ( i=0 ; i<N ; i++ ) {
    …
    … = bitStream[ i ];
    for ( j=0 ; j<M ; j++ ) {
        …
        … = qOutput[ j ];
        … = mBlock[ i ][ j ];
    }
    … = qOutput[ i ];
    … = ctab[…];
}
```

Fig.4. An example code with three different types of stream data structures

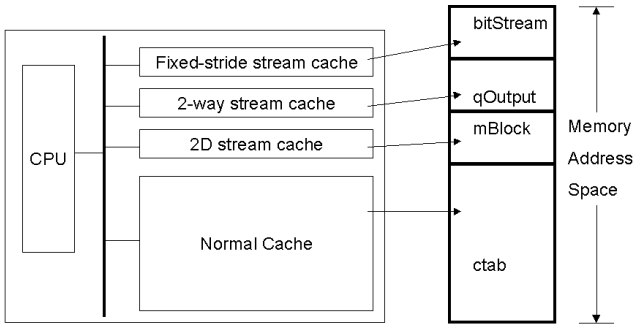


Fig.5. Example of stream cache assignment

IV. Overall Procedure

The overall procedure of the stream cache synthesis is depicted in Fig. 6. It consists of 5 main steps. The first step is to build the memory trace of applications. Memory trace contains physical access, symbolic name, size, and access type (read or write). We use the tracer of *ARMulator* to get physical addresses, sizes, and access types of the memory trace. To get a symbolic name we read the symbol table generated by the *armlink* and determine whether a physical address lies in any buffer region defined in the symbol table. The symbol table contains the start addresses of buffers and their sizes. We trace not only the physical addresses but also their symbolic names to identify the target stream data structures of the stream caches in the source code. Source code analysis can provide those information but it is rather complex and can be affected by the compiler.

The next step is the access pattern analysis step, the heart of the proposed methodology. For each predefined memory access pattern and the associated stream cache configuration, we make a specific cache simulation model. Assuming that a data structure has a regular memory access pattern we calculate the stride with first some accesses. Then, we simulate cache behaviors for each stream cache type. If the stream cache miss rate is below the threshold, the buffer is considered to fit for the corresponding stream cache. We set the threshold value to 3% determined after extensive simulations. If the correct stream cache is used, the cache miss rate is drastically reduced. During the simulation, we analyze interference between data structures and examine the possibility of stream cache sharing. If cache sharing does not increase the expected cache miss ratio, two data structures may share the same stream cache.

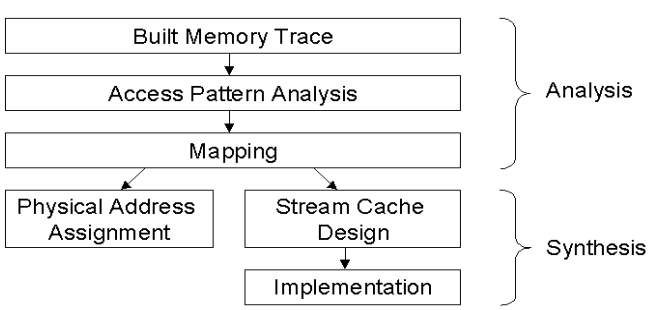


Fig.6. Overall procedure

The third step is to map the data structures to the stream cache modules. If a data structure has a fixed-stride access pattern and its simulated cache miss rate is below the threshold, it is mapped to a fixed-stride stream cache. We consider the most frequently accessed structure first. Fixed-stride stream cache has higher priority than other stream caches because fixed-stride stream cache has the least hardware area. Such heuristic works quite well as confirmed by experiments. Finding out the optimal mapping order is left for a future work. In this step, we consider the resource constraint given by the user such as the number of stream cache modules. For stream cache sharing, the stride values of two shared data structures should be the same and the interference may not increase the miss rate over a certain threshold.

The fourth step is to assign the target data structure to a data section determined from the previous mapping step. The address space is split into stream cache sections and a normal data cache section. Many embedded software development toolkits support this feature. We use the scatter loading feature of ARM Developer Suite to assign a buffer to a specific data section [12].

The last step is to synthesize the stream cache modules as decided in the mapping step. As explained earlier, for each stream cache module, we synthesize the necessary hardware glue logic as well as the control logic. If there is a stream cache with stride one and the line size is greater than one, it can use the burst mode of off-chip memory. In this case, the corresponding control logic should be augmented.

V. Experiment

A. *Setup*

We modified the tracer of ARMulator for trace-driven simulation and used Dinero IV 4.7 to simulate the normal data cache. We developed our own simulator for stream caches. The CPU model of ARMulator is ARM7TDMI and the number of stream caches is 8 at most for each application.

We used nine real multimedia applications. We first performed static analysis of the memory traces obtained from the simulation, and then cache simulation for performance comparison.

B. *Static Analysis Result*

Table 1 presents the number of data structures with regular memory access patterns for each application. For example, h263_enc has 14 buffers with fixed-stride access pattern, 1 buffer with 2-way and 18 buffers with 2D pattern. The examples of epic and gsm have no such data structure.

Table 2 presents the counts of actually mapped data structures and their portion among the total access counts. Since we restrict the maximum number of stream caches to 8, some data structures could not be mapped to a stream buffer, but to a normal cache.

TABLE I
Number of data structures with regular data access patterns

Application	Fixed-stride	2-way	2D
adpcm_dec	3	0	0
adpcm_enc	2	0	0
cd2dat	1	0	3
epic	0	0	0
gsm	0	0	0
h263_dec	12	0	4
h263_enc	14	1	18
jpeg	3	1	0

TABLE II
Number of mapped data structures within resource constraints

Application	Mapped data structures	Portion (%)	Used stream caches			
			Total	F	2w	2D
adpcm_dec	3	35.05	3	3		
adpcm_enc	2	21.82	2	2		
cd2dat	4	14.51	4	1		3
epic	0	0	0			
gsm	0	0	0			
h263_dec	15	8.91	8	6		2
h263_enc	22	33.13	8	2	1	5
jpeg	4	5.27	4	3	1	

In the cases of adpcm_dec, adpcm_enc, and h263_enc we can anticipate a significant performance improvement from the proposed architecture because mapped data structures take large portion of memory access counts. On the other hand, we may not expect such improvement from h263_dec and jpeg examples.

C. Cache Simulation Result

We compare the performance of the proposed architecture with two alternatives: conventional normal cache architecture, and the preloading scheme. When comparing the cache miss rate with preloading scheme, we ignore the cache pollution problem so that the performance of the preloading scheme is somewhat exaggerated. Cache pollution can be simulated if we use time-accurate simulation, which we avoid due to excessive simulation time.

Table 3 and 4 present simulation results for direct map and 2-way set associated normal caches respectively, varying the normal cache sizes. For a given cache size, the preloading and the stream cache architecture show significant improvement in terms of miss rates. As expected from the analysis step, the jpeg result is not so good though cache miss rate is reduced from the normal cache only. Therefore, it is not recommended to use stream caches for the jpeg application. A stream cache identifies the data structures with globally regular access pattern. But the preloading scheme detects also the case where only a part of accesses is regular. Unlike other cases the jpeg application has many partial regular access patterns, so the preloading scheme shows better result than the stream caches. In the cases of adpcm_dec, adpc_enc and h263_enc, the cache miss rates are reduced to less than half. In these cases it is favorable to use the stream caches.

TABLE III
Miss rate(%) with direct-map normal cache

Application		1K	2K	4K	8K	16K
adpcm_dec	Normal	14.58	8.13	3.43	3.15	3.03
	Preloading	13.16	7.61	3.27	2.99	2.87
	Stream	6.49	0.05	0.05	0.05	0.05
adpcm_enc	Normal	5.32	2.8	1.95	1.95	1.95
	Preloading	4.76	2.42	1.59	1.59	1.59
	Stream	0.19	0.13	0.10	0.08	0.07
cd2dat	Normal	9.01	3.70	2.44	1.79	1.79
	Preloading	5.01	2.05	1.10	0.76	0.76
	Stream	3.20	1.71	0.57	0.14	0.14
h263_dec	Normal	5.32	2.24	1.94	1.85	1.80
	Preloading	0.86	0.67	0.40	0.34	0.30
	Stream	3.32	0.45	0.16	0.09	0.02
h263_enc	Normal	8.23	5.71	4.91	3.96	3.45
	Preloading	2.29	1.82	1.22	0.71	0.48
	Stream	1.83	1.29	0.72	0.56	0.49
jpeg	Normal	4.45	1.55	0.73	0.64	0.45
	Preloading	3.81	1.09	0.28	0.23	0.05
	Stream	4.22	1.32	0.65	0.57	0.37

TABLE IV
Miss rate(%) with 2-way set associative normal cache

Application		1K	2K	4K	8K	16K
adpcm_dec	Normal	7.40	7.18	3.87	0.97	0.88
	Preloading	7.16	6.95	3.74	0.93	0.85
	Stream	0.09	0.05	0.05	0.05	0.05
adpcm_enc	Normal	3.17	1.76	0.14	0.14	0.14
	Preloading	3.07	1.88	0.11	0.11	0.11
	Stream	0.08	0.07	0.06	0.06	0.06
cd2dat	Normal	6.16	1.77	0.44	0.28	0.01
	Preloading	3.09	1.12	0.26	0.15	0.00
	Stream	2.55	0.86	0.14	0.14	0.14
h263_dec	Normal	2.07	1.62	1.21	1.12	1.09
	Preloading	0.71	0.44	0.14	0.08	0.06
	Stream	0.51	0.34	0.06	0.02	0.01
h263_enc	Normal	5.89	4.76	3.39	2.74	2.28
	Preloading	2.04	1.30	0.76	0.37	0.18
	Stream	1.50	1.07	0.64	0.50	0.48
jpeg	Normal	1.69	1.30	0.64	0.38	0.37
	Preloading	1.31	1.05	0.29	0.03	0.02
	Stream	1.65	1.27	0.60	0.34	0.34

Fig. 7 shows the minimum required cache size to have the cache miss rate less than 1%. Since the hardware overheads of the stream cache itself and its controller are much smaller than the normal cache size, we ignored the overheads in this comparison. Note that the preloading scheme needs comparable hardware overhead with the normal cache size as admitted in the reference [2]. The size 10K of Y coordinate means that any size of normal cache could not achieve such miss rate. Note that with stream caches, only 1K to 4K direct-map cache can meet such low cache miss rate constraint in all applications.

Using stream caches may increase the power consumption from increased off-chip memory accesses. But it will reduce the power consumption of on-chip cache accesses because the stream caches is much smaller than the normal cache. It should be verified in the future work, however, what would be the overall effect on power consumption.

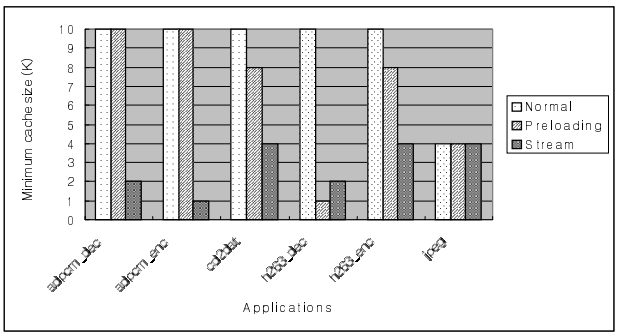


Fig.7. Minimum required direct-map cache size for less than 1% miss rate

D. Stream Cache Parameter

The effect of the number of lines is ignored in our experiments because it needs time accurate simulation to evaluate how much miss penalty of normal cache increases due to prefetching interference. Though we leave this experiment as a future work, we expect that the effect will be small enough to be ignored for miss rate computation.

On the other hand, the number of stream caches affects the performance. If more stream caches are used, more data structures will be mapped to stream caches, thus reducing the miss rate more. The performance improvement should be obtained with the increased hardware cost. Table 5 shows such trade-off relationship.

In h263_dec, the total access count portion is not varied much so that using 4 stream caches would have the best performance/cost ratio. In the case of h263_enc the portion of using 4 stream caches is much smaller than that of using 8 stream caches. Fig. 8 presents the result of miss rate simulation of each case graphically.

TABLE V
Mapped data structures according to the count of stream cache used

Application	Mapped data structures	Portion (%)	Used stream caches			
			Total	F	2w	2D
h263_dec	11	8.81	4	2		2
	15	8.91	8	6		2
	16	9.04	12	6		6
h263_enc	6	22.51	4	3	1	
	22	33.13	8	2	1	5
	23	33.16	12	3	1	8

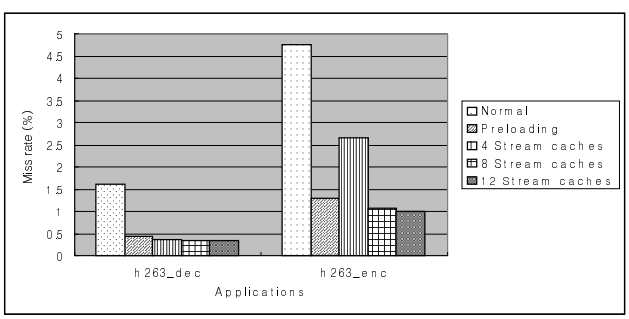


Fig. 8. Miss rate with 2K 2-way associative cache according to the count of stream caches used

VI. Conclusions

This paper presents a static analysis technique of memory access patterns based on the physical memory traces. And, we introduce a new cache module, called stream cache, that prefetches the data with higher accuracy without complicated hardware prediction logic. Experiments with real multimedia applications show very promising results that stream cache is a useful memory module for memory system exploration of multimedia embedded systems. Our work can be extended to more diverse optimization possibilities. We can increase the number of regular memory access patterns and add more diverse configurations optimized for access patterns. Not only customized cache but also communication channel or bus architecture could be the target of our framework.

Data layout optimizations for stream cache would be helpful. While the compiler helps us only to assign buffers to specific data section in current framework, if the compiler can be aware of the stream cache, new compiler optimization may greatly improve the performance. Also, source code optimization would affect the performance with the stream cache architecture.

References

[1] P. Baglietto, M. Maresca and M.Migliardi, "Image processing on high-performance RISC systems," Proc. of the IEEE, vol 84, no.7, 1996.

[2] J.L. Baer and T.F Chen, "An effective on-chip preloading scheme to reduce data access penalty," Proc. of the Conference on Supercomputing, 1991.

[3] T.F, Chen and J. L. Baer, "Effective hardware-based data prefetching for high-performance processors," IEEE Trans. on Computers. VOL 44, No. 5, May 1995.

[4] T.F. Chen and J.L. Baer, "A performance study of software and hardware prefetching schemes," Proc. of the 21st Annual International Symposium on Computer Architecture, pp. 223-232, 1994.

[5] P.Grun, N.Dutt, and A.Nicolau. "APEX: access pattern based memory architecture exploration," in ISSS, 2001.

[6] S. Rubin, R.Bodik, and T.Chilimbi. "An efficient profile-analysis framework for data-layout optimizations," in POPL 2002.

[7] N.P.Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in ISCA 1990.

[8] S.Palacharla and R.E.Kessler, "Evaluating stream buffers as a secondary cache replacement," in ISCA 1994.

[9] S. Przybylski, "Sorting out the new DRAMs," in Hot Chips Tutorial, Stanford, CA, 1997.

[10] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, Custom Memory Management Methodology, Kluwer, 1998.

[11] C. Zhang and S.A. McKee, "Hardware-only stream prefetching and dynamic access ordering," in ICS 2000.

[12] ARM Ltd., Linker and Utilities Guide, ARM Developer Suite Release 1.2, 2001.