A Novel Memory Structure for Embedded Systems: Flexible Sequential and Random Access Memory

Ying Chen[1], Karthik Ranganathan[2], Vasudev V Pai[3], David J. Lilja[1], and Kia Bazargan[1]

[1]Department of Electrical and Computer Engineering, University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455, USA

[2]Data Communications Division, Cypress Semiconductor Corp.,
198 Champion Ct., San Jose, CA 95134, USA

[3] Marvell Semiconductor, 5400 Bayfront Plaza, Mailstop E-201,
Santa Clara, CA 95054, USA

{wildfire, lilja, kia}@umn.edu; kr@cypress.com; pvasudevus@yahoo.com

Abstract. The on-chip memory performance of embedded systems directly affects the system designers' decision about how to allocate expensive silicon area. We investigated a novel memory architecture, *flexible sequential and random access memory* (FSRAM), for embedded systems. To realize sequential accesses, small "links" are added to each row in the RAM array to point to the next row to be prefetched. The potential cache pollution is ameliorated by a small *sequential access buffer* (SAB). To evaluate the architecture-level performance of FSRAM, we ran the Mediabench benchmark programs [1] on a modified version of the Simplescalar simulator [2]. Our results show that the FSRAM improves the performance of a baseline processor with a 16KB data cache up to 55%, with an average of 9%; furthermore, the FSRAM reduces 53.1% of the data cache miss count on average due to its prefetching effect. We also designed RTL and SPICE models of the FSRAM [3], which show that the FSRAM significantly improves memory access time, while reducing power consumption, with negligible area overhead.

Keywords: on-chip memory, Sequential Access Buffer (SAB), Media benchmark, Flexible Sequential and Random Access Memory (FSRAM)

1 Introduction

Rapid advances in high-performance computing architectures and semiconductor technologies have drawn considerable interest to high performance memories. Increases in hardware capabilities have led to performance bottlenecks due to the time required to access the memory. Furthermore, the on-chip memory performance in embedded systems directly affects designers' decisions about how to allocate expensive silicon area. Off-chip memory power consumption has become the energy consumption bottleneck as embedded applications become more data-centric.

Most of the recent research has tended to focus on improving performance and power consumption of on-chip memory structures [4, 5, 6] rather than off-chip memory. Moon *et al* [7] investigated a low-power sequential access on-chip memory designed to exploit the numerous sequential access patterns in digital signal processing (DSP) applications. Prefetching techniques from traditional computer architecture have also been used to enhance on-chip memory performance for embedded systems [8, 9, 10]. Other studies have investigated energy efficient off-chip memory for embedded systems, such as automatic data migration for multi-bank memory systems [11].

None of these previous studies, however, have investigated using off-chip memory structures to improve on-chip memory performance. This study demonstrates the performance potential of a novel, low-power, off-chip memory structure, which we call the *flexible sequential and random access memory* (FSRAM), to support flexible memory access patterns. In addition to normal random access, the FSRAM uses an extra "link" structure, which bypasses the row decoder, for sequential accesses. The link structure reduces power

* Supported in part by the Minnesota Supercomputing Institute.

consumption and decreases memory access times; moreover, it aggressively prefetches data into the on-chip memory. In order to eliminate the potential data cache pollution caused by prefetching, a small fully associative *sequential access buffer* (SAB) is used in parallel with the data cache. VHDL and HSPICE models of the FSRAM have been developed to evaluate its effectiveness at the circuit level. Embedded multimedia applications are simulated to demonstrate its performance potential at the architecture level. Our results show significant performance improvement with little extra area used by the link structures.

The remainder of this paper is organized as follows. Section 2 introduces and explains the FSRAM and the SAB. In Section 3, the experimental setup is described. The architecture level performance analysis and area, timing and power consumption evaluations of the FSRAM are presented in Section 4. Section 5 discusses related work. Finally, Section 6 summarizes and concludes.

2 Flexible Sequential Access Memory

Our flexible sequential and random access memory (FSRAM) architecture is an extension of the sequential memory architecture developed by Moon, *et al* [7]. They argued that since many DSP applications have static and highly predictable memory traces, row address decoders can be eliminated. As a result, memory access would be sequential with data accesses determined at compile time. They showed considerable power savings at the circuit level.

While preserving the power reduction property, our work extends their work in two ways: (1) in addition to circuit-level simulations, we perform architectural-level simulations to assess the performance benefits at the application level; and (2) we extend the sequential access mechanism using a novel enhancement that increases sequential access flexibility.

2.1 Structure of the FSRAM

Fig. 1 shows the basic structure of our proposed FSRAM. There are two address decoders to allow simultaneous read and write accesses[1]. The read address decoder is shared by both the memory and the “link” structure. However, the same structure is used as the write decoder for the link structure, while the read/write decoder is required only for the memory. As can be seen, each memory word is associated with a link structure, an OR gate, a multiplexer, and a sequencer.

The link structure indicates which successor memory word to access when the memory is being used in the sequential access mode. With 2 bits, the link can point to four unique successor memory word lines (e.g., N+1, N+2, N+4, and N+8). This link structure is similar to the "next" pointer in a linked-list data structure. Note that Moon et al [7] hardwired the sequencer cell of each row to the row below it. By allowing more flexibility, and the ability to dynamically modify the link destination, the row address decoder can be bypassed for many more memory accesses than previous mechanisms to provide greater potential speedup.

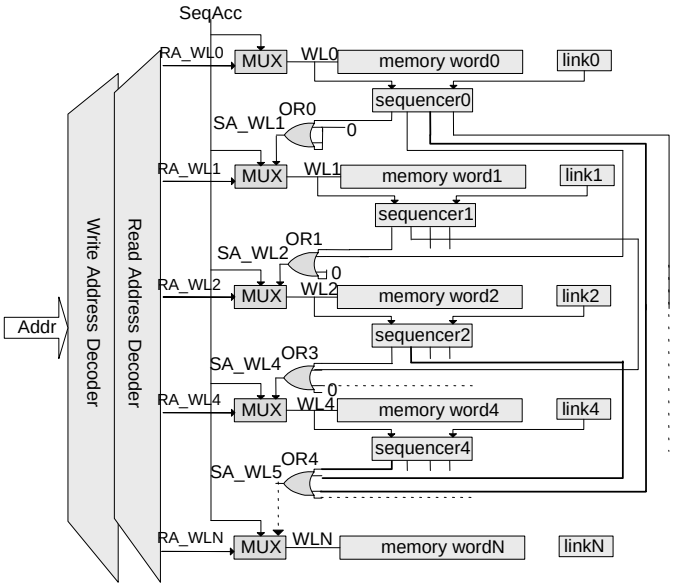


Fig. 1. The FSRAM adds a link, an OR gate, a multiplexer, and a sequencer to each memory word**.**

The OR block shown in Fig. 1 is used to generate the sequential address. If any of the four inputs to the OR block is high, the sequential access address (SA_WL) will be high (Fig. 2.a). Depending on the access mode signal (SeqAcc), the multiplexers choose between the row address decoder and the sequential cells. The role of the sequencer is to determine the next sequential address according to the value of the link (Fig. 2.b). If WL is high, then one of the four outputs is high. However if *reset* is high, then all four outputs go low irrespective of WL.

The timing diagram is shown in figure 3. At the start of a new cycle the OR gate generates the sequential address while the random address is generated by the address decoder. The MUX chooses one of the two addresses. The value of the word line is stored in the sequencer while write is still high. Once write goes low, the word line is pulled down. (All word lines are

[1] Throughout the paper, all experiments are performed assuming dual-port memories. It is important to note that our FSAM does not *require* the memory to have two ports. The reason we chose two ports is that most modern memory architectures have multiple ports to improve memory latency.

automatically pulled low when write bar is asserted by means of a pull down transistor.) During the period while write is low, the sequencer uses the stored word line value and the link value to determine whether any of its outputs should be asserted. This output is now fed to the OR gates. At the start of the next cycle, the process is repeated once again.

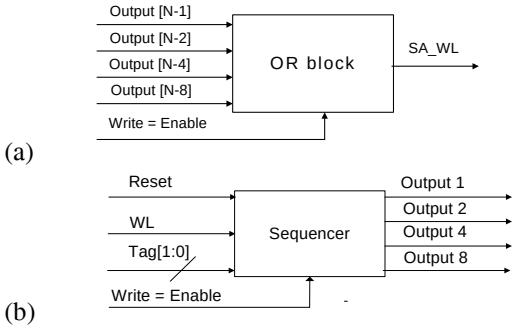


Fig. 2. (a) Block diagram of the OR block, (b) block diagram of the sequencer.

Figure 3 shows an example in which a data value of 0 is stored in row 0, and a value of 1 is stored in row 2. The steps involved are:

1. First, row 0 is to be addressed, hence RA/SAbar is to be kept high after address is given to the decoder.
2. This signal selects the output of the MUX and activates the memory line.
3. The data bit is written and WRITE goes low, hence WL0 follows.
4. During WRITE low, Pre1Pre0 is made 01 to select the next alternate location, which is number 2.
5. When WRITE goes high next time, the OR gate output is enabled and out02 is sent out into input of MUX at row 2. Since WRITE is kept high, MUX gives out WL2.
6. This combined with high WRITE writes the bit into the location.
7. Steps 3 through 7 repeat for continued access.

Read access can be done similarly. To summarize, a block of data can be read by first accessing the first element using the row address decoder, and then following the links in the sequential mode. The next subsection discusses how and when the link values are actually written.

The area overhead of the FSRAM consists of four parts - the link, OR gate, multiplexer, and sequencer. The overhead is in about the order of 3-7% of the total memory area for the word line size of 32 bytes and 64 bytes. More detailed area overhead results are shown in Table 3 in Section 4.2.

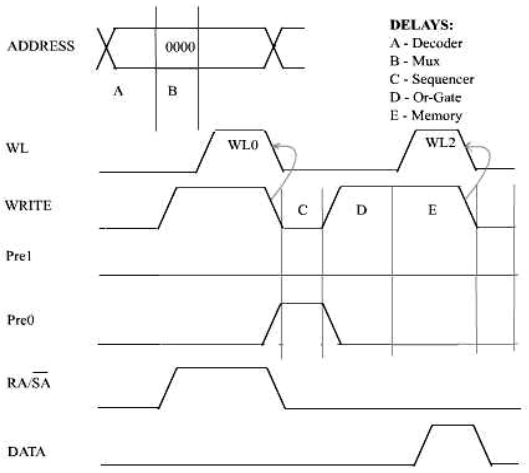


Fig 3. Timing diagram of a memory word cell. Delay of the random write is: A + B + E, Delay of the sequential write is : C + D + E

2.2 Update of the Link Structure

The link associated with each off-chip memory word line is dynamically updated using data cache miss trace information and run-time reconfiguration of the sequential access target. In this manner, the sequentially accessed data blocks are linked when compulsory misses occur. Since the read decoder for the memory is the same physical structure as the write decoder for the link structure, the link can be updated in parallel with a memory access. The default link value of the link is 0, which actually means the next line (2^0=1).

We note that the read and write operations to the memory data elements and the link RAM cells can be done independently. The word lines can be used to activate both the links and the data RAM cells for read or write (not all of the control signals are shown in Fig. 1).

There are a number of options for writing the link values:

1. The links can be computed at compile-time and loaded into the data memory while instructions are being loaded into the instruction memory.
2. The link of one row could be written while the data from another row is being read.
3. The link can be updated while the data of the same row is being read or written.

Option 1 is the least flexible approach since it exploits only static information. However, it could eliminate some control circuitry that supports the runtime updating of the links. Options 2 and 3 update the link structure at run-time and so that both exploit dynamic run-time information. Option 2, however, needs more run-time data access information compared to Option 3

and thus requires more control logic. We decided to examine Option 3 in this paper since the dynamic configuration of the links can help in subsequent prefetches.

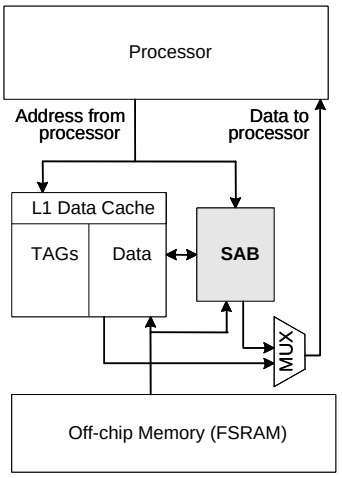


Fig. 4. The placement of the Sequential Access Buffer (SAB) in the memory hierarchy.

2.3 Accessing the FSRAM and the SAB

In order to eliminate potential cache pollution caused by the prefetching effect of the FSRAM, we use a small fully associative cache structure, which we call the *Sequential Access Buffer* (SAB). In our experiments, the on-chip data cache and the SAB are accessed in parallel, as shown in Fig.4. The data access operation is summarized in Fig. 5.

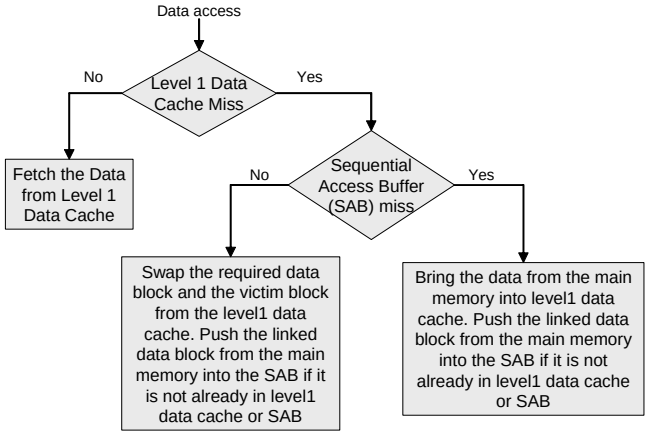


Fig. 5. Flowchart of a data access when using the SAB and the FSRAM

When a memory reference misses in both the data cache and the SAB, the required block is fetched into the data cache from the off-chip memory through random access mode. Furthermore, a data block pointed to by the link of the data word being currently read is pushed into the SAB through sequential access mode if it is not already in the on-chip memory. That is, the link is followed and the result is stored in the SAB. When a memory reference misses in the data cache but hits in the SAB, the required block and the victim block in the data cache are swapped. Additionally, the data block linked to the required data block, but not already in on-chip memory, is pushed into the SAB. In this data access scheme whenever a data block is accessed through the link structure, it is sequential access mode, otherwise, random access mode.

3 Experimental Methodology

To evaluate the system level performance of the FSRAM, we used SimpleScalar 3.0 [2] to run the Mediabench [1] benchmarks using this new memory structure. The basic processor configurations are based on Intel Xscale [12], The Intel XScale microarchitecture is a RISC core that can be combined with peripherals to provide applications specific standard products (ASSP) targeted at selected market segments. The basic processor configurations are as the following: 32 KB instruction L1 caches with 32-byte data lines, 2-way associativity and 1 cycle latency, no L2 cache, and 50 cycle main memory access latency. The default SAB size is 8 entries. The machine can issue two instructions per cycle. It has a 32-entry load/store queue and one integer unit, one floating point unit, and one multiplication / division unit, with 1 cycle, 1 cycle, 2 cycles, and 12 cycles latency respectively. The branch predictor is bimodal and has 128 entries. The instruction and data TLBs are fully associative and have 32 entries. The link structure in the off-chip memory was simulated by using a large enough table to hold both the miss addresses and their link values. The link values are updated by monitoring the L1 data cache miss trace. Whenever the gap between two continuous misses is 1x, 2x, 3x, 4x block line size, we update the link value correlated to the memory line that causes the first miss in the two continuous misses.

3.1 Benchmark Programs

We used the Mediabench [13] benchmarks ported to the SimpleScalar simulator for the architecture-level simulations of the FSRAM. We used four of the benchmark programs, *adpcm, epic, g721* and *mesa*, in this study because they were the only ones that worked with the Simplescalar PISA instruction set architecture. These four benchmarks represent the applications for audio coding, image compression, voice compression, and 3-D graphic library respectively.

Since the FSRAM link structure links successor memory word lines (Section 3.1), we show the counts of

the address gap distances between two consecutive data cache misses in Table 1. We see from these results that the address gap distances of 32, 64, 128, 256 and 512 bytes are the most common, while the other address gap distances occur more randomly. Therefore, the FSRAM evaluated in this study supports address gap distances of 32, 64, 128 and 256 bytes for a 32-byte cache line, while distances of 64, 128, 256 and 512 bytes are supported for a 64-byte cache line.

For all of the benchmark programs tested, the dominant gap distances are between 32 and 128 bytes. Most of the tested benchmarks, except *g721*, have various gap distances distributed among 32 to 256 bytes. When the gap increases to 512 bytes, *epic* and *mesa* still exhibit similar access patterns while *adpcm* and *g721* have no repeating patterns at this gap distance. Another important issue for the evaluation of benchmark program performance is the overall memory footprint estimated from the cache miss rates. Table 2 shows the change in the L1 data cache miss rates for the baseline architecture as the size of the data cache is changed. In general, these benchmarks have small memory footprints, especially *adpcm* and *g721*. Therefore, we chose data cache sizes in these simulations to approximately match the performance that would be observed with larger caches in real systems. The default data cache configuration throughout this study is 16 KB with a 32-byte line and 2-way set associativity.

3.2 Processor Configurations

The following processor configurations are simulated to determine the performance impact of adding an FSRAM to the processor and the additional performance enhancement that can be attributed to the SAB.

orig: This is the baseline architecture with no link structure in the off-chip memory and no prefetching mechanism.

FSRAM: This configuration is described in detail in Section 3.1. To summarize, this configuration incorporates a link structure in the off-chip memory to exploit sequential data accesses.

Table 1. The frequencies (counts) of the various address distance gaps between two consecutive data cache misses for the tested benchmark programs

	Adpcm encode	adpcm decode	epic encode	epic decode	G721 encode	G721 decode	Mesa mipmap	Mesa osdemo	Mesa Texgen
32Bytes	121	121	167	82	609512	590181	78740	2212	229004
64 Bytes	7157	7157	3552	43	93	94	9	50896	22809
128 Bytes	979	979	1864	80	0	0	5	497	13441
256 Bytes	3237	3237	36	392	0	0	14	9	2
512 Bytes	0	0	5	896	0	0	3	1	16457

Table 2. The L1 data cache miss rates for the baseline architecture with various L1 cache sizes

	adpcm encode	adpam decode	epic encode	epic decode	g721 encode	g721 decode	mesa mipmap	Mesa Osdemo	Mesa Texgen
2KB	0.0214	0.0174	0.1424	0.1248	0.0010	0.0013	0.0894	0.0207	0.0735
4KB	0.001	0.0011	0.0703	0.0612	0.0003	0.0004	0.0444	0.0173	0.0337
8KB	0.0011	0.0011	0.0362	0.0591	0.0001	0.0001	0.0176	0.0142	0.0127
16KB	0.0010	0.0010	0.0162	0.0569	0.0000	0.0000	0.0086	0.0123	0.0068
32KB	0.0010	0.0010	0.0150	0.0535	0.0000	0.0000	0.0059	0.0112	0.0048

FSRAM_SAB: This configuration uses the FSRAM with an additional small, fully associative SAB in parallel with the L1 data cache. The details of the SAB were given in Section 3.3

tnlp: This configuration adds tagged next line prefetching [14] to the baseline architecture. With tagged next line prefetching, a prefetch operation is initiated on a miss and on the first hit to a previously prefetched block. Tagged next line prefetching has been shown to be more effective than prefetching only on a miss [15]. We use this configuration to compare against the prefetching ability of the *FSRAM*.

tnlp_PB: This configuration enhances the *tnlp* configuration with a small, fully associative Prefetch Buffer (PB) in parallel with the L1 data cache to eliminate the potential cache pollution caused by next line prefetching. We use this configuration to compare against the prefetching ability of the *FSRAM_SAB* configuration.

4 Performance Evaluation

In this section we evaluate the performance of an embedded processor with the FSRAM and the SAB by analyzing the sensitivity of the processor configuration *FSRAM_SAB* as the on-chip data cache parameters are varied. We also show the timing, area, and power consumption results based on RTL and SPICE models of the FSRAM.

4.1 Architecture-level Performance

We first examine the *FSRAM_SAB* performance compared to the other processor configurations to show the data prefetching effect provided by the FSRAM and the cache pollution elimination effect provided by the SAB. Since the FSRAM improves the overall performance by improving the performance of the on-chip data cache, we evaluate the *FSRAM_SAB* performance while varying the values for different data cache parameters including the cache size, associativity, block size, and the SAB size. We also analyze both the prefetching effect of the FSRAM and its overhead.

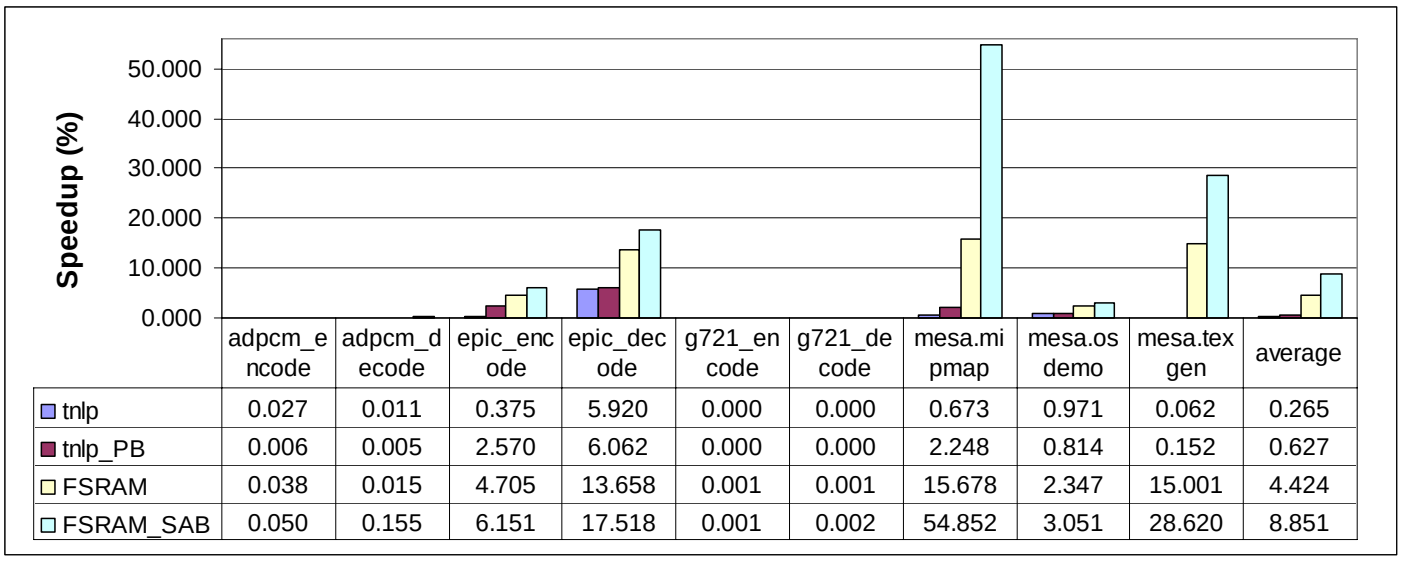
Throughout Section 4.1, the average statistics are calculated using the execution time weighted average of all of the benchmarks [16].

4.1.1 Performance Improvement due to FSRAM

To show the performance obtained from the FSRAM and the SAB, we compare the relative speedup obtained by all four processor configurations described in Section 3.2 (i.e., *tnlp, tnlp_PB, FSRAM, FSRAM_SAB*) against the baseline processor configuration (*orig*). All of the processor configurations use a 16 KB L1 data cache with a 32-byte data block size and 2-way set associativity.

As shown in Fig. 6, the *FSRAM* configuration produces an average speedup of slightly more than 4% over the baseline configuration compared to a speedup of less than 1% for *tnlp*. Adding a small prefetch buffer (PB) to the *tnlp* configuration (*tnlp_PB)* improves the performance by about 1% compared to the *tnlp* configuration without the prefetch buffer. Adding the same size SAB to the FSRAM configuration (*FSRAM_SAB*) improves the performance compared to the FSRAM without the SAB by an additional 8.5%. These speedups are due to the extra small cache structures that eliminate the potential cache pollution caused by prefetching directly into the L1 cache. Furthermore, we see that the FSRAM without the SAB outperforms tagged next-line prefetching both with and without the prefetch buffer. The speedup of the FSRAM with the SAB compared to the baseline configuration is 8.5% on average and can be as high as 54% (*mesa_mipmap*).

Benchmark programs *adpcm* and *g721* have very small performance improvements, because their memory footprints are so small that there are very few data cache misses to eliminate in a 16KB data cache (Table 2) Nevertheless, from the statistics shown in Fig. 6, we can still see *adpcm* and *g721* follow the similar performance trend described above. These small improvements could be system noises. The reason why we keep the benchmarks *adpcm* and *g721* is because sometimes they have performance improvements, which are due to the fast sequential accesses, as shown in section 4.1.2



	adpcm_encode	adpcm_decode	epic_encode	epic_decode	g721_encode	g721_decode	mesa.mipmap	mesa.osdemo	mesa.texgen	average
tnlp	0.027	0.011	0.375	5.920	0.000	0.000	0.673	0.971	0.062	0.265
tnlp_PB	0.006	0.005	2.570	6.062	0.000	0.000	2.248	0.814	0.152	0.627
FSRAM	0.038	0.015	4.705	13.658	0.001	0.001	15.678	2.347	15.001	4.424
FSRAM_SAB	0.050	0.155	6.151	17.518	0.001	0.002	54.852	3.051	28.620	8.851

Fig. 6. Relative speedups obtained by the different processor configurations. The baseline is the original processor configuration. All of the processor configurations use a 16KB data L1 cache with 32-byte block and 2-way associativity.

4.1.2 Parameter Sensitivity Analysis

We are interested in the performance of FSRAM with different on-chip data caches to exam how the off-chip FSRAM main memory structure improves on-chip memory performance. So in this section we study the effects of various data cache sizes (i.e., 2KB, 4KB, 8KB, 16KB, 32KB), data cache associativities (i.e., 1way, 2way, 4way, 8way), cache block sizes (i.e., 32 bytes, 64 bytes) and the SAB sizes (i.e., 4 entries, 8entries, 16entreis) on the performance. The baseline processor configuration through this section is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

The Effect of Data Cache Size. Fig. 7 shows the relative speedup distribution among orig, *tnlp_PB* and *FSRAM_SAB* for various L1 data cache sizes (i.e., 2KB, 4KB, 8KB, 16KB, 32KB). The total relative speedup is *FSRAM_SAB* with a L1 data cache size over the baseline, which is orig with a 2KB L1 data. Each bar in Fig. 7 is divided into three parts because the relative speedup is the accumulation of three contributions: the relative speedup attributed to *orig* with a L1 data cache size configuration over the baseline; the relative speedup attributed to *tnlp_PB* with a L1 data cache size configuration over the baseline; the relative speedup of *FSRAM_SAB* with a L1 data cache size configuration over the baseline. Therefore the gray part of each bar is the relative speedup attributed to *orig*, the dark gray part of each bar is the relative speedup attributed to *tnlp_PB* over that attributed to *orig*, and the while part of each bar is the relative speedup attributed to *FSRAM_SAB* over that attributed to *tnlp_PB*.

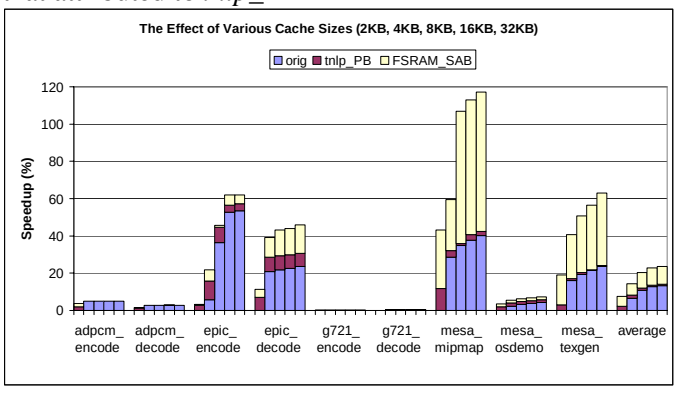


Fig. 7. Relative speedups distribution among the different processor configurations (i.e., *orig, tnlp_PB, FSRAM_SAB*) with various L1 data cache sizes (i.e., 2KB, 4KB, 8KB, 16KB, 32KB). The baseline is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

As shown, with the increase of L1 data cache size the relative speedup of *tnlp_PB* over *orig* decreases. FSRAM_SAB, in contrast, constantly keeps speedup on top of *tnlp_PB* across the different L1 data cache sizes. Furthermore, *FSRAM_SAB* even outperforms *tnlp_PB* with a larger size L1 data cache for most of the cases and on average. For instance, *FSRAM* with a 8KB L1 data cache outperforms *tnlp_PB* with a 32KB L1 data cache. However, *tnlp_PB* only outperforms the baseline processor with a bigger size data cache for *epic_decode* and *mesa_osdemo*.

The improvement in the performance can be attributed to several factors. While the baseline processor does not perform any prefetching, the tagged next line prefetching prefetches only the next word line. The fact that our method can prefetch with strides is one contributing factor in the smaller memory access time. Furthermore, prefetching is realized using sequential access, which is faster than random access. Another benefit is that prefetching with different strides does not require an extra large table to store the next address to be accessed.

tnlp_PB and *FSRAM_SAB* improve performance in the case that the performance of *orig* increases with the increase of L1 data cache size. However, they have little effect in the case that the performance of *orig* increases with the increase of L1 data cache size, which means the benchmark program has a small memory footprint (i.e., *adpcm, g721*). For adpcm, *tnlp* and *FSRAM_SAB* still improve performance when the L1 data cache size is 2K. For g721, the performance almost keeps the same all the time due to the small memory footprint.

The Effect of Data Cache Associativity. Fig. 8 shows the relative speedup distribution among orig, *tnlp_PB* and *FSRAM_SAB* for various L1 data cache associativity (i.e., 1way, 2way, 4way, 8way).

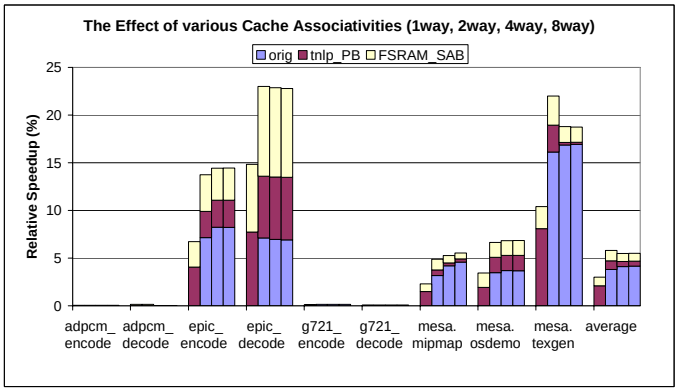


Fig. 8. Relative speedups distribution among the different processor configurations (i.e., *orig, tnlp_PB, FSRAM_SAB*) with various L1 data cache associativity (i.e., 1way, 2way, 4way, 8way). The baseline is the original processor configuration with a 16KB data L1 cache with 32-byte block size and 2-way associativity.

As known, increasing the L1 data cache associativity typically reduces the number of L1 data cache misses.

The reduction in misses reduces the effect of prefetching from *tnlp_PB* and *FSRAM_SAB*. As can be seen, the performance speed up of *tnlp_PB* on top of *orig* decreases as the L1 data cache associativity increases. The speed up almost disappears when the associativity is increased to 8way for mesa_mipmap and mesa_texgen. However, *FSRAM_SAB* still provides significant speedups.

tnlp_PB and *FSRAM_SAB* still have little impact on the performance of *adpcm* and *g721* because of their small memory footprints.

The Effect of Data Cache Block Size. Fig. 9 shows the relative speedup distribution among *orig, tnlp_PB* and *FSRAM_SAB* for various L1 data cache block sizes (i.e., 32B, 64B).

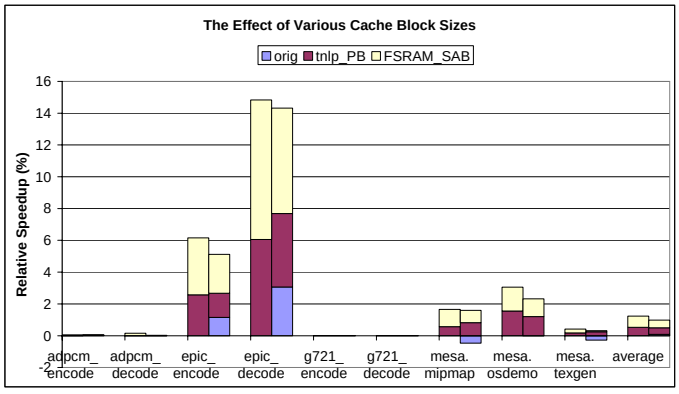


Fig. 9. Relative speedups distribution among the different processor configurations (i.e*., orig, tnlp_PB, FSRAM_SAB*) with various L1 data cache block sizes (i.e., 32B, 64B). The baseline is the original processor configuration with a 16KB data L1 cache with 32-byte block size and 2-way associativity.

As known increasing the L1 data cache block size typically reduces the number of L1 data cache misses. For all of the benchmarks the reduction in misses reduces the effect of prefetching from *tnlp_PB* and *FSRAM_SAB*. As can be seen, the performance speed up of *tnlp_PB* on top of *orig* decreases as the L1 data cache block size increases from 32-bytes to 64 bytes. However, the increasing of the L1 data cache block size can also cause potential pollutions as for *epic_encode* and *mesa_mipmap*. Tnlp with a small prefetching buffer reduces the pollution, and FSRAM_SAB further speeds up the performance.

The Effect of SAB Size. Fig. 10 shows the relative speedup distribution among orig, tnlp_PB and FSRAM_SAB for various SAB sizes (i.e., 4 entries, 8 entries, 16 entries).

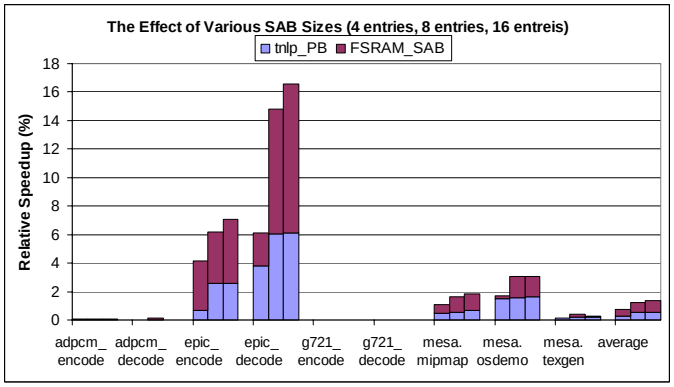


Fig. 10. Relative speedup distribution among the different processor configurations (i.e*., tnlp_PB, FSRAM_SAB*) with various SAB sizes (i.e., 4 entries, 8 entries, 16 entries). The baseline is the original processor configuration with a 16KB data L1 cache with 32-byte block size and 2-way associativity.

Fig. 10 compares the FSRAM_SAB approach to a tagged next-line prefetching that uses the prefetch buffer that is the same size as SAB. As shown, FSRAM_SAB always add speedup on top of tnlp_PB. Further, *FSRAM_SAB* outperforms *tnlp* with a bigger size prefetch buffer. This result indicates that FSRAM_SAB is actually a more efficient prefetching mechanism than a traditional tagged next-line prefetching mechanism.

tnlp_PB and *FSRAM_SAB* still have little impact on the performance of adpcm and g721 because their small memory footprints.

4.1.3 The Prefetching Effect of the FSRAM

To evaluate the data prefetching mechanism of FSRAM, in Fig 11, we show the percentage of the data blocks fetched from the off-chip memory FSRAM to the on-chip data cache that turn out to be useful. As we can see for most benchmark programs useful data blocks are prefetched from the gaps of 32 bytes, 64 bytes, and 128 bytes. On average 34%, 31% and 33.5% prefetched data blocks are useful from the three gaps respectively. Useful data blocks fetched from the gap of 256 bytes only exists for epic_decode. g721_encode, g721_decode, and mesa.mipmap do not have useful data blocks fetched from the gaps of 128 bytes or 256 bytes because very small amount of counts of these two address gaps exist for the three benchmark programs (Table 1).

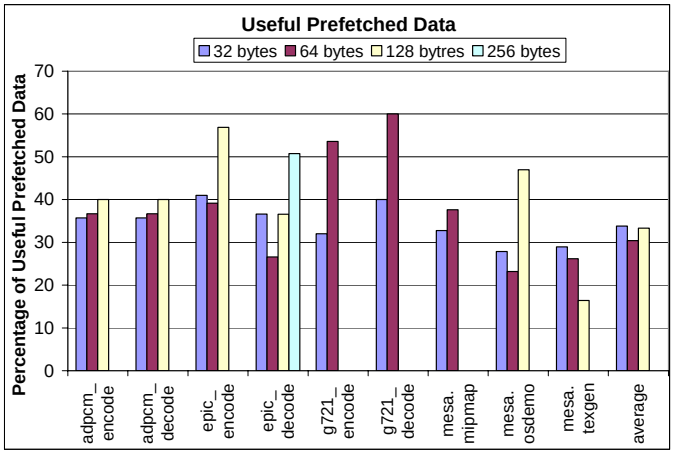


Fig. 11. The percentage of the prefetched data blocks (with the gap of 32 bytes, 64 bytes, 128 bytes, 256 bytes) that turned out to be useful. The baseline is the original processor configuration with a 16KB data L1 cache with 32-byte block size and 2-way associativity, and an 8-entry SAB.

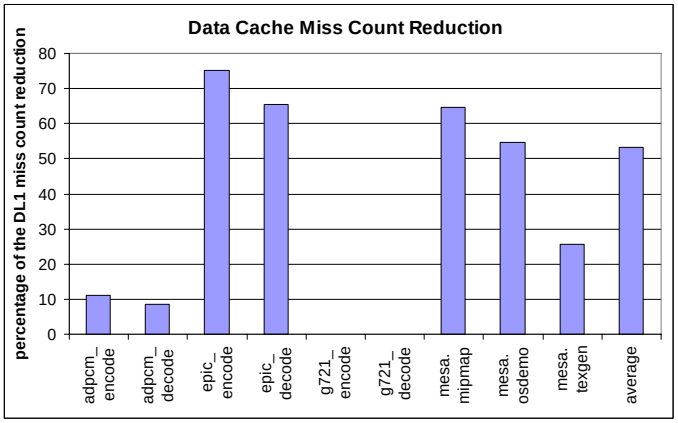


Fig. 12. The percentage of the data cache miss count reduction due to the prefetching effect of FSRAM. The baseline is the original processor configuration with a 16KB data L1 cache with 32-byte block size and 2-way associativity, and an 8-entry SAB.

The prefetching mechanism of FSRAM brings useful data blocks and thus reduces the data cache miss count as shown in Fig. 12. Most of the benchmark programs have a significant amount data cache miss count reduced. The reduction is up to 75% (i.e., epic_encode), and on average is 53.1%. The miss count reductions for g721_encode and g721_decode are zero because the memory footprint of these two programs are so small that the miss count is zero with the 16 KB data cache (Table 2). As a result, the FSRAM has efficient prefetching effect.

4.1.4 The Overhead Due to the FSRAM_SAB

The overhead of the FSRAM is shown in Fig. 13 in the term of increases in memory traffic between the off-chip memory FSRAM and the on-chip data cache. The memory traffic contains both instruction and data cache misses, and the prefetched data traffic, which are the memory access counts.

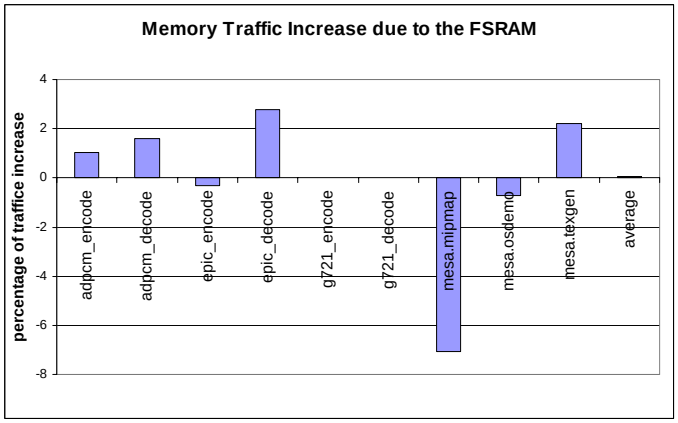


Fig. 13. Increases in memory traffic. The baseline is the original processor configuration with a 16KB data L1 cache with 32-byte block size and 2-way associativity, and an 8-entry SAB.

As shown most benchmark programs have a small amount of memory traffic increase that is less than 3%. mesa.mipmap, epic_encode, and mesa.osdemo actually have decreased memory traffic because the amount of data cache miss counts is larger than the extra traffic caused by the prefetching mechanism. For mesa.mipmap the memory traffic is even reduced by 7%, which is the reason for the large performance speedup of 54% for mesa.mipmap in Fig 6. On average 0.1% more memory traffic is caused by the prefetching effect of the FSRAM, which is negligible.

The overall speedup of the benchmark program, which is discussed in section 4.1.1., is the comprehensive result from both the prefetching effect (Fig. 11. and Fig. 12.) of the FSRAM and it's overhead (Fig. 13.)

4.2 Timing, area and Power Consumption

We implemented the FSRAM architecture in VHDL to verify its functional correctness at the RTL level. We successfully tested various read/write combinations of row data *vs.* links. Depending on application requirements, one or two decoders can be provided so that the FSAM structure can be used as a dual-port or single-port memory structure. In all our experiments, we assumed dual-port memories since modern memory structures have multiple ports to decrease memory latency.

In addition to the RTL level design, we implemented a small 8x8 (8 rows, 8 bits per row) FSRAM in HSPICE using 0.18µm technology to test timing correctness and evaluate the delay of sequencer blocks. Note that unlike the decoder, the sequencer block's delay is independent

of the size of the memory structure: it only depends on how many rows it links to (in our case: 4).

By adding sequencer cells, we will be adding to the area of the memory structure. However, in this section we show that the area overhead is not large, especially considering the fact that in today's RAMs, a large number of memory bits are arranged in a row. An estimate of the percentage increase in area was calculated using the formula $(\frac{A1}{A1-A2}-1)x100\%$ where A1 = Total Area and A2 = area occupied by the link, OR gate, MUX and the sequencer. Table 3 shows the results of the increases in area for different memory row sizes. The sequencer has two SRAM bits, which is not many compared to the number of bits packed in a row of the memory. We can see that the sequencer cell logic does not occupy a significant area either.

Table 3. Area overhead of FSRAM with various memory word line sizes

No. of bits per row of memory	Increase in area due to the MUX and the sequencer
8 (1 byte)	216%
16 (2 bytes)	119%
64 (8 bytes)	23.0%
256 (32 bytes)	7.12%
512 (64 bytes)	3.10%

As can be seen, the percentage increase in area drops substantially as the number of bits in each word line increases. Hence the area overhead is almost negligible for large memory blocks.

Using the HSPICE model, we compared the delay of the sequencer cell to the delay of a decoder. Furthermore, by scaling the capacity of the bit lines, we estimated the read/write delay and hence, calculated an overall speedup of 15% of sequential access compared to random access.

Furthermore, the power saving is 16% in sequential access at VDD = 3.3v in the 0.18 micron CMOS HSPICE model.

5 Related Work

The research related to this work can be classified into three categories: on-chip memory optimizations, off-chip memory optimizations, and hardware-supported prefetching techniques.

In their papers, Panda *et. al.* [4, 5] address data cache size and number of processor cycles as performance metrics for on-chip memory optimization. Shiue *et al.* [6] extend this work to include energy consumption and show that it is not enough to consider only memory size increase and miss rate reduction for performance optimization of on-chip memory because the power consumption actually increases. In order to reduce power consumption, Moon *et al.* [7] designed an on-chip sequential access only memory specifically for DSP applications that demonstrates the low-power potential of sequential access.

A few papers have addressed the issue of off-chip memory optimization, especially power optimization, in embedded systems. In a multi-bank memory system Dela Luz *et al.* [11] show promising power consumption reduction by using an automatic data migration strategy to co-locate the arrays with temporal affinity in a small set of memory banks. But their approach has major overhead due to extra time spent in data migration and extra power spent to copy data from bank to bank.

Zucker *et al.* [10] compared hardware prefeching techniques adopted from general-purpose applications to multimedia applications. They studied a stride prediction table associated with PC (program counter). A data-cache miss-address-based stride prefetching mechanism for multimedia applications is proposed by Dela Luz *et al.* [11]. D. Joseph and D. Grunwald described a prefetching mechanism to identify previously resident lines to a level-one cache, called the Markov predictor [17]. A table was used to store the probability in a Markov chain. D. M. Koppelman [18] proposed a prefetching scheme for multiprocessors using instruction history, called neighborhood prefetching. All these studies show promising results at the cost of extra on-chip memory devoted to a table structure of non-negligible size. Although low-cost hybrid data prefetching slightly outperforms hardware prefetching, it limits the code that could benefit from prefetching [9]. Sbeyti *et. al.* [8] propose an adaptive prefetching mechanism which exploits both the miss stride and miss interval information of the memory access behavior of only MPEG4 in embedded systems.

Unlike previous approaches, we propose a novel off-chip memory with little area overhead (3-7% for 32 bytes and 64 bytes data block line) and significant performance improvements, compared to previous works that propose expensive on-chip memory structures. Our study investigated off-chip memory structure to improve on-chip memory performance, thus leaves flexibility for designer's to allocate expensive on-chip silicon area. Furthermore, we improved power consumption of off-chip memory.

6 Conclusions

In this study, we proposed the FSRAM mechanism that makes it possible to eliminate the use of address decoders during sequential accesses and also random accesses to a certain extent.

We find that FSRAM can efficiently prefetch the linked data block into on-chip data cache and improve performance by 4.42% on average for an embedded

system using 16KB data cache. In order to eliminate the potential cache pollution caused by the prefetching, we used a small fully associative cache called SAB. The experiments show FSRAM can further improve the tested benchmark programs performances to 8.85% on average using the SAB. Compared to the tagged next-line prefetching, FSRAM_SAB constantly performs better and can still speedup performance when tnlp_PB cannot. This indicates that FSRAM_SAB is a more efficient prefetching mechanism. Furthermore, FSRAM_SAB reduces 53.1% of the data cache miss count on average with a negligible amount of extra memory traffic introduced by its prefetching mechanism. Although the data cache miss counts for most benchmarks are small, FSRAM_SAB is still able to improve performance because the sequential access itself is faster than random access, which proves the efficiency of the sequential access.

FSRAM has both sequential accesses and random accesses. With the expense of negligible area overhead (3-7% for 32 bytes and 64 bytes data block line) from the link structure, we obtained a speedup of 15% of sequential access over random access from our designed RTL and SPICE models of FSRAM. Our design also shows that sequential access save 16% power consumption.

The link structure/configuration explored in this paper is not the only way; a multitude of other configurations can be used. Depending upon the requirement of an embedded application, a customized scheme can be adopted whose level of flexibility during accesses best suits the application. For this, prior knowledge of access patterns within the application is needed. In the future, it would be useful to explore power-speed trade-offs that may bring about a net optimization in the architecture.

References

[1] C.Lee, M. Potkonjak, and W. H. Mangione-Smith. "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems", In Proc. of the 30th Annual International Symposium on Microarchitecture (Micro 30), December 1997

[2] Doug Burger and Todd M. Austin. "The simplescalar tool set version 2.0", Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[3] Ying Chen, Karthik Ranganathan, Amit Puthenveetil, Kia Bazargan, and David J. Lilja, “FSRAM: Flexible Sequential and Random Access Memory for Embedded Systems”, Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 04-01, February, 2004.

[4] P. R. Panda, N. D. Dutt, and A. Nicolau. “Data cache sizing for embedded processor applications”, Technical Report TCS-TR-97-31, University of California, Irvine, June 1997.

[5] P. R. Panda, N. D. Dutt, and A. Nicolau. “Architectural exploration and optimization of local memory in embedded systems”, International Symposium on System Synthesis (ISSS 97), Antwerp, Sept. 1997.

[6] W. Shiue, C. Chakrabati, “Memory Exploration for Low Power Embedded Systems”, IEEE/ACM Proc.of 36th. Design Automation Conference (DAC'99), June 1999.

[7] J. Moon, W. C. Athas, P. A. Beerel, J. T. Draper, “Low-Power Sequential Access Memory Design”, IEEE 2002 Custom Integrated Circuits Conference, pp.741-744, Jun 2002.

[8] H. Sbeyti, S. Niar, L. Eeckhout, "Adaptive Prefetching for Multimedia Applications in Embedded Systems", DATE'04, EDA IEEE, 16-18 february 2004,Paris, France

[9] A. D. Pimentel, L. O. Hertzberger, P. Struik, P. Wolf, "Hardware versus Hybrid Data Prefetching in Multimedia Processors: A Case Study", in the Proc. of the IEEE Int. Performance, Computing and Communications Conference (IPCCC 2000), pp. 525-531, Phoenix, USA, Feb. 2000

[10] D. F. Zucker, M. J. Flynn, R. B. Lee, "A Comparison of Hardware Prefetching Techniques For Multimedia Benchmarks", In Proceedings of the International Conferences on Multimedia Computing and Systems, Himshima, Japan, June 1996

[11] V. De La Luz, M. Kandemir, I. Kolcu, "Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems", DAC, pp 213-218, 2002

[12] Intel corporation, "The intel XScale Microarchitecture technical summary", http://www.intel.com/design/intelxscale/xscaledatasheet4.htm, Technical report, 2001

[13] http://www.cse.psu.edu/~mdl/mediabench.tar.gz

[14] J. E. Smith, W. C. Hsu, “Prefetching in Supercomputer Instruction Caches”, In proceedings of Supercomputing92, pp. 588-597, 1992

[15] S. P. VanderWiel and D. J. Lilja, “Data Prefetch Mechanisms”, ACM Computing Surveys, Vol. 32, Issue 2, June 2000, pp. 174-199

[16] D. J. Lilja, “Measuring Computer Performance”, Cambridge University Press, 2000

[17] D. Joseph and D. Grunwald, “Prefetching using Markov Predictors”, in Proceedings of the International Symposium on Computer Architecture, June 1997, pp. 252-263.

[18] D. M. Koppelman, “Neighborhood Prefetching On Multiprocessors Using Instruction History”, International Conference on Parallel Architectures and Compilation Techniques, October 2000, pp. 123-132

Ying Chen received her PhD and MS degree in Electrical Engineering from the University of Minnesota and the BS degree in Electrical Engineering from Tsinghua University. Starting fall of 2005, she will work as an Assistant Professor at San Francisco State University. Her main research interests include hardware verification methodology, multiprocessors, multithreaded architectures, memory systems, and reconfigurable computing. She is a student member of the IEEE.

Karthik Ranganathan received his Bachelors degree in Electronics Engineering from the University of Mumbai, India. He was an ECE Department Fellow as well as a Graduate Teaching Assistant at the University of Minnesota. He has interned with the VLSI division at Seagate Technologies and with the Applications Engineering group at Cypress Semiconductor. Subsequently, he graduated with a Master's Degree in Electrical Engineering from the University of Minnesota. He has been a Product/Test Engineer with the Data Communications division at Cypress Semiconductor for about 2 years. He is currently a Sr. Product Engineer working with the 'CCD' division of Cypress Semiconductor Corp.

Vasudev Pai received the B.E. (Hons.) degree in electrical and electronics engineering from Birla Institute of Technology and Science, Pilani, India and the M.S. degree in electrical engineering from the University of Minnesota, Twin Cities in 2000 and 2005, respectively. He was with IBM Corporation from 2000 to 2002 where he worked on the hardware verification of the Power4 processor at Austin, TX. He has also held internships at Texas Instruments and Cypress Semiconductor. Currently, he is working as a digital design engineer at Marvell Semiconductor, Santa Clara, CA.

David J. Lilja received the Ph.D. and M.S. degrees, both in Electrical Engineering, from the University of Illinois at Urbana-Champaign, and a B.S. in Computer Engineering from Iowa State University in Ames. He currently is a Professor and Head of Electrical and Computer Engineering, and a Fellow of the Minnesota Supercomputing Institute, at the University of Minnesota in Minneapolis. He also serves as a member of the graduate faculties in Computer Science and Scientific Computation. He has been a visiting senior engineer in the Hardware Performance Analysis group at IBM in Rochester, Minnesota, and a visiting professor at the University of Western Australia in Perth. Previously, he worked as a research assistant at the Center for Supercomputing Research and Development at the University of Illinois, and as a development engineer at Tandem Computers Incorporated (now a division of Hewlett-Packard) in Cupertino, California. He has chaired and served on the program committees of numerous conferences; was a distinguished visitor of the IEEE Computer Society; is a member of the IEEE and the ACM; and is a registered Professional Engineer in Electrical Engineering in Minnesota and California. His primary research interests are in high-performance computer architecture, parallel computing, hardware-software interactions, nano-computing, and performance analysis.

Kia Bazargan received his Bachelors degree in Computer Science from Sharif University in Tehran, Iran, and his M.S. and PhD in Electrical and Computer Engineering from Northwestern University in Evanston, IL in 1998 and 2000 respectively. He is currently an Assistant Professor in the Electrical and Computer Engineering at the University of Minnesota. He has served on the technical program committee of a number of IEEE sponsored conferences (e.g., ISPD, ICCAD, ASP-DAC, GLSVLSI). He was a guest co-editor of ACM Transactions on Embedded Computing Systems (ACM TECS), Special Issue on Dynamically Adaptable Embedded Systems in 2003. He is an Associate Editor of IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems. He was a recipient of NSF CAREER award in 2004. His research interests are computer-aided design, FPGAs and reconfigurable computing.