In Proceedings of the International Symposium on Rapid System Prototyping, pages 137-144, Paris, France, June 2009.

High-performance buffer mapping to exploit DRAM concurrency in multiprocessor DSP systems

Dongwon Lee[1], Shuvra S. Bhattacharyya[2], and Wayne Wolf[1]
[1]School of ECE, Georgia Institute of Technology, Atlanta, Georgia 30332, USA
[2]Department of ECE, University of Maryland, College Park, Maryland 20742, USA
dwlee@gatech.edu, ssb@umd.edu, and wolf@ece.gatech.edu

Abstract

Design methodologies and tools based on the synchronous dataflow (SDF) model of computation have proven useful for rapid prototyping and implementation of digital signal processing (DSP) applications on multiprocessor systems. One significant problem that arises when mapping applications onto such embedded multiprocessors is the memory wall problem, which is becoming increasingly dominant in multiprocessor environments. In this paper, to help alleviate the memory wall problem, we propose a novel, high-performance buffer mapping policy for SDF-represented DSP applications on multiprocessor systems that support the shared-memory programming model. The proposed policy exploits the bank concurrency of a DRAM main memory system according to the analysis of the major forms of parallelism. The throughput is measured on both synthetic and real benchmarks. The simulation results show that the proposed buffer mapping policy is very useful, especially in memory-intensive applications where the total execution time of computational tasks is relatively small compared to that of memory operations. The performance improvement produced by our method is generally attained at the cost of additional banks and decreased bank utilization.

1 Introduction

Multiprocessor systems are widely used in both general-purpose and embedded computing to keep up with the Moore's performance law. In spite of the performance advantages of multiprocessors, the memory wall problem becomes increasingly severe in multiprocessor environments due to communication and synchronization overhead. A hierarchical memory system including caches and DRAM main memory concurrency can alleviate the memory wall problem; however effective analysis and utilization of such memory systems are challenging. In this paper, we study the latter form of memory system enhancement – use of DRAM concurrency – for increasing the performance of multiprocessor digital signal processing (DSP) systems.

A dataflow graph is very natural representation for DSP applications [1]. Synchronous dataflow (SDF) is a special case of dataflow, in which the number of tokens consumed or produced by each actor (computational task) in each firing (task execution) is known a priori [2]. An important class of DSP applications exhibits this form of synchrony, which allows compilers to perform more powerful scheduling and buffer mapping compared to more general models of computation (e.g., see [3, 4]). This results in reduced run-time overhead, streamlined buffer memory requirements, and more predictable run-time behavior.

In our proposed buffer mapping methodology, we first represent the targeted DSP application as a SDF graph. We then apply task-level scheduling, which includes three stages – assigning actors to processors, ordering the execution of actors on each processor, and specifying the firing times of actors. Which of these stages are done at compile time depends on the scheduling strategy [5]. Since it is possible to get the good estimates of actor execution times in DSP applications, the self-timed scheduling strategy is often attractive. In self-timed scheduling, actor assignment and execution ordering are performed at compile time, while the exact firing times of actors are determined at run-time. Mechanisms for run-time synchronization are needed in self-timed scheduling because the memory transaction order can in general be changed during execution [1].

An inter-processor communication (IPC) graph is then generated from the SDF application graph and the self-timed scheduling result. The IPC graph models self-timed execution of the given application based on the given schedule, and explicitly shows the requirements of inter-processor communication and synchronization [1]. In the shared-memory programming model, communication is implicitly done by memory read and write transactions, but synchronization requires an explicit mechanism. In embedded computing systems, the software synchronization mechanism based on hardware primitives, which is widely used in general-purpose computing systems, is not desirable because of its hardware cost. Instead, bounded buffer synchronization (BBS) and unbounded buffer synchronization (UBS) protocols are considered [6], which use the shared main memory for synchronization as well as communication.

The BBS protocol is more attractive because of its bounded buffer feature.

In the BBS protocol, a part of DRAM main memory should be allocated to each IPC edge for communication and synchronization. In this paper, we define the logical dataflow graph buffer corresponding to an IPC edge *e* as the *IPC buffer* associated with *e*, and we define the allocation of physical memory space to the IPC buffer as the *buffer mapping* associated with the edge. Owing to the synchrony (statically-known rates of data production and consumption) of SDF-based DSP applications, IPC buffer mapping for such applications can be done at compile-time. Such compile time buffer mapping can be performed in various ways, such as sequential mapping, randomly distributed mapping, etc.

In this paper, an efficient buffer mapping policy is proposed for SDF-based DSP applications. This buffer mapping policy is targeted specifically for multiprocessor systems that support the shared-memory programming model. The proposed policy emphasizes efficient exploitation of DRAM resource concurrency for application performance enhancement.

This paper is organized as follows. Section 2 describes the previous work and motivation. In Section 3, the BBS protocol and the buffer mapping problem are described in detail. In Section 4, properties of contemporary DRAM main memory systems are described. Section 5 presents our high-performance buffer mapping policy. We compare the throughput of our proposed policy with the conventional sequential mapping policy in Section 6.

2 Previous work and motivation

Buffer mapping has a big impact on system performance. To understand this impact, it is useful to view DRAM main memory system as a finite state machine whose next state is determined by the current state and the incoming memory operation command. One factor that complicates memory analysis is that DRAM main memory exhibits non-uniform access latency depending on the access history.

Previous related work on SDF techniques computes or measures processor throughput considering a zero- or uniform-latency DRAM main memory system [2, 6]. Furthermore, the impact of different IPC buffer mapping policies on performance has not been considered much. In general-purpose computing, most of the research on improving memory performance has focused on the memory controller techniques such as scheduling and memory address interleaving [7, 8, 9].

In this paper, an efficient buffer mapping policy, which can be carried out by a high-level compiler, is proposed to take into account the non-uniform access latency of contemporary DRAM systems. Due to the high level of abstraction at which we apply this analysis and the critical role of the memory system in determining overall system performance, our method is useful to incorporate during rapid prototyping and design space exploration.

3 BBS synchronization protocol and buffer mapping

In this section, the BBS protocol and related buffer mapping considerations are described in detail. Fig. 1 depicts a SDF application graph and an associated scheduling result on three processors. Note that this scheduling result is in general not unique – it is determined by one of many scheduling possibilities [1]; the one illustrated here is a represented result that we have chosen for the purpose of illustration. Every actor is indexed with a unique number and this number is used to identify the actor in the schedule. The scheduling result shown here is derived by using the classic HLFET algorithm [10] under the self-timed paradigm. In this paper, homogeneous SDF (HSDF) [2] is considered. Since all arbitrary SDF graphs can be converted into equivalent HSDF graphs [2], the techniques of this paper are also applicable to general SDF graphs, as long as the SDF-to-HSDF transformation is applied appropriately as a preprocessing step. From the SDF application graph and the schedule, we obtain the IPC graph shown in Fig. 2. In the IPC graph, each white circle represents a computation actor, and each gray circle represents a communication actor. Communication actors are labeled as (S) or (R) to represent inter-processor *send* and *receive* operations, respectively. The actors that are assigned to the same processor by the given schedule are connected together so that they form a cycle. The ordering of actors along each of these cycles corresponds to the actor ordering for the corresponding processor in the self-timed schedule. Each of these cycles represents the iterative, sequential execution of the subset of actors that is assigned to a given processor.

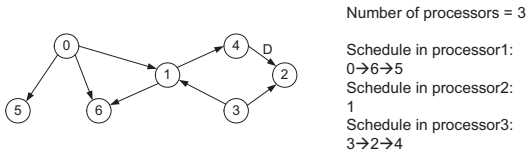

Figure 1. SDF Application graph and its schedule

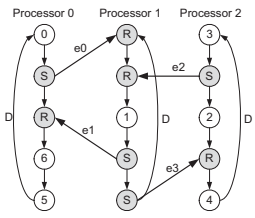

Figure 2. IPC graph

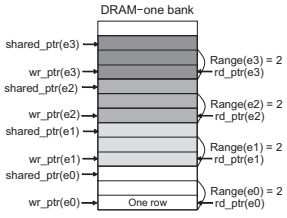

Figure 3. Sequential buffer mapping

For each edge in the SDF application graph whose source and sink actors are assigned to different processors by the given schedule, IPC edge is instantiated between the associated send and receive actors in the IPC graph. For example, there are four IPC edges, *e*0-*e*3 in Fig. 2. For further details on IPC graph construction, we refer the reader to [1].

A feedback edge is an edge that is contained in at least one cyclic path (equivalently, it is part of a strongly connected component). When analyzing IPC graphs, it is useful to identify *feedback edges* in the graphs in advance because the buffer sizes of such edges are bounded, and the associated buffer size bounds can be derived through low-complexity analysis of the IPC graph [1]. Furthermore, through a transformation called convert-to-SC graph, the IPC graph can be converted into a form such that all IPC edges are feedback edges, and the original application behavior is preserved. In the remainder of this paper, we assume that such a transformation is applied so that all IPC edges are feedback edges, and therefore have statically-known buffer size bounds.

In conjunction with providing buffer bounds, IPC edges that are feedback edges can be implemented with an efficient synchronization protocol called *bounded buffer synchronization* (BBS). Intuitively, BBS needs only to provide for buffer-empty checking at run-time, and buffer-overflow checking can be avoided entirely by simply allocating a block of physical memory to the buffer that is equal to the statically-computed buffer size bound of the edge.

Suppose that we are given an IPC edge *e* which is a feedback edge, and let src(*e*) and snk(*e*) denote, respectively, the source and sink actors of *e.* Fig. 3 depicts the sequential buffer mapping for the IPC graph of Fig. 2 under the BBS protocol. For simplicity, we suppose the transferred data size is one row. In the BBS protocol [6], a write pointer wr_ptr(*e*) for *e* is maintained on the processor that executes src(*e*), a read pointer rd_ptr(*e*) for *e* is maintained on the processor that executes snk(*e*), and a copy of wr_ptr(*e*) is maintained in some shared memory location shared_ptr(*e*). The pointers rd_ptr(*e*) and wr_ptr(*e*) are initialized to zero and delay(*e*), respectively. Just after each execution of src(*e*), the new data value produced onto *e* is written into the shared memory buffer for *e* at offset wr_ptr(*e*) and is updated by the following operation: wr_ptr(*e*) ← (wr_ptr(*e*) + 1) mod range(*e*). shared_ptr(*e*) is updated to contain the new value of wr_ptr(*e*). Just before each execution of snk(*e*), the value contained in shared_ptr(*e*) is repeatedly examined until it is found to be not equal to rd_ptr(*e*). Then, the data value residing at offset rd_ptr(*e*) of the shared memory buffer for *e* is read, and rd_ptr(*e*) is updated by the operation rd_ptr(*e*) ← (rd_ptr(*e*) + 1) mod range(*e*). Since all IPC edges in our proposed methodology can be assumed to be feedback edge, the size of each buffer is known at compile time, so we can determine the start address and range of each buffer at compile time. Only the modulo-based increase of the pointers needs to be carried out during run-time.

The sequential buffer mapping of Fig. 3 is the most straightforward, but does not utilize very useful features of contemporary DRAM main memory systems.

4 Contemporary DRAM main memory system

In this section, contemporary DRAM main memory system is described in terms of its useful features. A contemporary DRAM main memory system including memory controller is shown in Fig. 4. The DRAM main memory system consists of several *ranks*, each rank has several *chips*, and each chip has several *internal banks*. The multiple internal banks within a chip and the several ranks provide multiple levels of concurrency. Note, however, that the multiple chips within a rank are not for providing concurrency, but for providing wide data transfer. In the example of Fig. 4, the number of ranks, the number of chips, and the number of internal banks are all four. In this paper, a single-channel SDRAM main memory system is considered, where all banks and ranks share a single address/command bus and single data bus as shown in Fig. 4. Furthermore, we assume that a commodity SDRAM chip (e.g., see [11, 12]) is used as a component, i.e. each bank capacity and the number of banks per chip are fixed, but we can adjust the numbers of chips and ranks according to the configuration.

A DRAM memory system has two useful features: bank concurrency and page mode. The data accesses to different banks or data accesses to the open row result in low access latency. The first is to utilize the bank concurrency and the second is to utilize the page mode. First, the bank concurrency makes it possible to hide the latency caused by precharge and row activation commands. Throughout this paper, we use the terms *command* and *transaction* when describing main memory operation. A command is issued by the memory controller to the DRAM. Examples of operations referenced in a command are precharge, row activation,

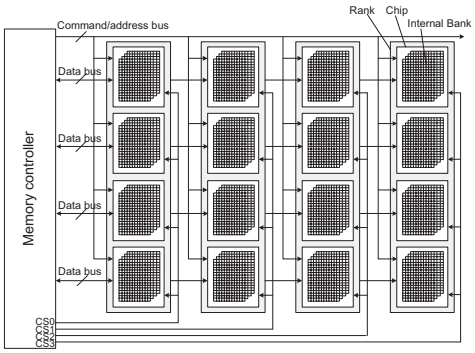

Figure 4. Contemporary DRAM main memory system

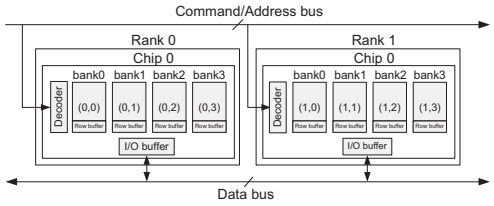

Figure 5. Example: DRAM main memory system

and column access. On the other hand, a transaction refers to an interaction between the processor and the memory controller. Examples of transactions are load and store operations. In general, a single transaction generates one or more commands depending on the row buffer management policy and the transaction schedule. Since the address/command bus and data bus are shared among all banks in a single-channel memory system, parallel execution of commands is actually done in a pipelined fashion. Second, the page mode is used to exploit temporal and spatial row locality of DRAM memory accesses. Each internal bank has its own row buffer. When the current memory transaction goes to the same row as the previous memory transaction, the current transaction gets the data from the open row buffer, not from the bank itself.

Several techniques exist to reorder memory transactions in memory controller in order to utilize both bank concurrency and page mode [8, 13, 14]. We focus in this paper on exploiting one of them – bank concurrency – carefully, and on managing our exploitation systematically through a high-level compiler. In order to analyze the pure effect of bank concurrency, we need to eliminate the effect of page mode on the system performance. Therefore, the close page policy is selected as the row buffer management policy in this paper. Under the close page policy, every transaction is converted to a sequence of precharge - row activation - column access commands regardless of row hit/miss, so we can thereby eliminate the effect of page mode on the system performance.

If we use a multi-rank configuration as in Fig. 4, rank concurrency is also used in addition to bank concurrency. In this paper, the term *bank concurrency* indicates the concurrency among the banks within each chip. The term *rank concurrency* indicates the concurrency among the banks across the rank. In some cases, it is useful to prioritize these forms of concurrency. For example, suppose that the application needs five banks. Since we assume that we are using a commodity DRAM chip with four internal banks as a component, two ranks are required and total of eight banks are available, as shown in Fig. 5. For simplicity, we suppose that each rank consists of one DRAM chip in this example. We need to select five banks out of the available eight banks. There are two categories of selections: bank concurrency first or rank concurrency first. For example, selecting (0,0), (0,1), (0,2), (0,3), and (1,0) gives bank concurrency a higher priority but selecting (0,0), (1,0), (0,1), (1,1), and (0,2) gives

Table 1. Resource utilization of commands

		Pre-charge	Row activation	Column access
Bank-level resource	Row, column, row buffer	O	O	O
Rank-level resource	Decoder	O	O	O
	I/O buffer			O
System-level resource	Cmd/addr bus	O	O	O
	Data bus			O

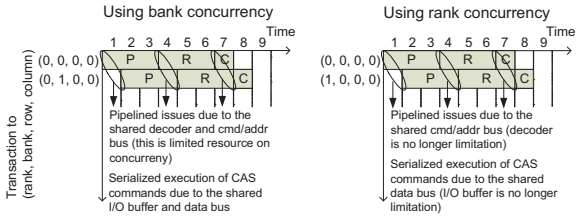

Figure 6. Transaction execution in single-channel memory system

rank concurrency a higher priority, where (i, j) indicates the j^{th} bank of the i^{th} rank.

In order to solve this selection problem, we should consider two points: resources that limit concurrency and data bus turn-around time. To see the effect of limited resources on performance when using bank concurrency or rank concurrency, consider the sequence of two memory transactions shown in Fig. 6. To make the execution diagram, we need information on the resource utilization of commands, as shown in Table 1. In Fig. 6, each transaction is represented by (rank, bank, row, column) and we suppose that precharge (P) and row activation (R) commands take three cycles and column access (C) command takes one cycle to complete. As shown in Fig. 6, in a single-channel memory system, the two transactions take eight cycles to complete in the both cases. This is because both forms of concurrency are equally limited by the single command/address bus and the single data bus. For reference, consider the same transactions in a multi-channel memory system where each rank has its own command/address bus and data bus. In this environment, there is no limit on the rank concurrency, but the bank concurrency is still limited by rank-level resources, such as the decoder and I/O buffer. This difference results in the better performance of rank concurrency in a multi-channel environment. Second, when using rank concurrency, the bus turn-around time should be considered. This rank-to-rank switching overhead is zero in SDRAM [15].

In summary, in a single-channel SDRAM main memory system, using the bank concurrency and using the rank concurrency show no performance difference. Therefore, in this paper, we randomly select which form of concurrency is firstly used.

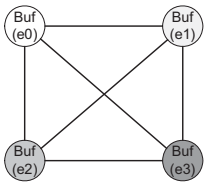

Figure 7. Interference graph of IPC buffers

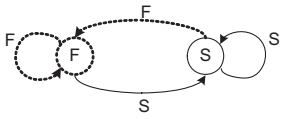

Figure 8. 1-bit predictor of sync_read result

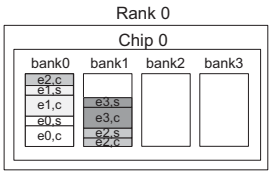

Figure 9(a). Sequential buffer mapping

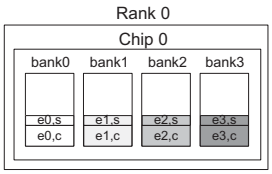

Figure 9(b). Balanced buffer mapping at edge-level (even1)

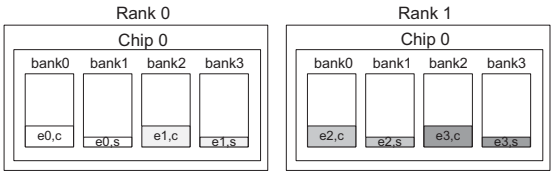

Figure 9(c). Balanced buffer mapping at comm/sync-level (even2)

5 Buffer mapping policy to exploit DRAM concurrency

In this section, we propose a high-performance buffer mapping policy to exploit bank concurrency and rank concurrency if a multi-rank configuration is used. Note that our methods focus on balanced mapping of IPC buffers, not on balanced access. Even though the balanced mapping is a prerequisite for the balanced access, balanced mapping does not guarantee the balanced access. It can be more difficult to achieve balanced access in a system using the BBS protocol. In the BBS protocol, both communication and synchronization are performed by memory transactions to the shared main memory. If a specific receive actor tries a sync_read before the corresponding send actor completes its associated write and sync_write, then the sync_read check fails, and it must be retried until the sync_read succeeds. This generally generates more accesses to the specific IPC buffer corresponding to the IPC edge that has the synchronization check failure. So in systems using the BBS protocol, the balanced access is strongly affected by the rate of synchronization failures and the associated need to reattempt the synchronizations. Techniques for reducing the rate of synchronization failures are not covered in this paper; this is a useful direction for further investigation that may provide further benefits on top of the techniques that we propose in this paper.

We propose a high-performance buffer mapping policy to utilize DRAM concurrency by analyzing two kinds of parallelism: IPC edge-level parallelism and comm/sync-level parallelism within each IPC edge. First, in order to analyze the edge-level parallelism, an interference graph of the IPC edges is obtained by using a graph coloring technique. In the interference graph, each vertex represents an IPC buffer and each edge between vertices represents interference of the associated buffers. That is, two vertices are connected with an edge if the corresponding two buffers are accessed in overlapping segments of time. So, to facilitate exploitation of parallelism, the two buffers should be mapped to different banks. The interference graph of the IPC graph of Fig. 2 is shown in Fig. 7. According to interference analysis based on many simulations for this example, all of four buffers turn out to exhibit interference, so the four buffers should all be mapped to different banks.

In order to exploit DRAM concurrency more, we use an additional form of parallelism at the comm/sync-level within each IPC edge. Every IPC buffer consists of two kinds of transaction buffers: a communication transaction buffer and a synchronization transaction buffer. In balanced mapping using the edge-level parallelism, the communication buffer and the synchronization buffer of an IPC buffer are mapped to the same bank. In contrast, in balanced mapping using comm/sync-level parallelism, the communication and synchronization buffers are mapped to the different banks. To analyze comm/sync-level parallelism, a 1-bit state machine as shown in Fig. 8 is used to predict the results of sync_read operations. In Fig. 8, 'F' and 'S' mean "failure" and "success," respectively; the circles indicate the prediction results; the arrows indicate the actual observed results. The 1-bit predictor toggles the prediction result when the prediction result and the real result are different – misprediction occurred. Based on the prediction result, we use a speculative read scheme, which is different from a general conservative read scheme. In the conservative scheme, whether the read transaction is issued or not is determined only after checking the result of the corresponding sync_read. In contrast, in the speculative read scheme, the read transaction can be issued at the same cycle as the corresponding sync_read if the prediction result is 'S'.

This speculative read scheme, however, does not always work well. If the misprediction rate is high, useless read transactions can waste memory bandwidth and this overhead may overshadow the benefits provided by parallelism. Thus, we should determine whether or not the comm/sync-level parallelism is used based on the prediction accuracy which can be estimated at compile time.

To show the difference of our proposed methods compared to conventional sequential mapping, two types of balanced mappings are illustrated in Figure 9(b) and (c). Figure 9(b) shows a balanced mapping using edge-level parallelism (even1), and Figure 9(c) shows a balanced mapping using

comm/sync-level parallelism on top of edge-level parallelism (even2). These buffer mapping examples pertain to the IPC graph of Fig. 2. Here, e*i*,c and e*i*,s represent the communication transaction buffer and the synchronization transaction buffer, respectively, for IPC edge e*i*.

To provide scalability, we use a modulo-operation to perform the mapping of buffers to banks. If the application requires a number of buffers that exceeds the maximum number of configurable DRAM banks, then buffer mapping is done based on modulo-*B* operation, where *B* is the total number of available banks.

Any performance improvement of our proposed buffer mapping policy over sequential mapping comes at the cost of the additional banks and bank under-utilization. This kind of cost/benefit analysis is useful to perform in conjunction with rapid prototyping and SoC-based design space exploration.

6 Simulation results and analysis

In this section, application throughput is measured for the sequential, even1 (balanced, edge-level), and even2 (balanced, comm/sync-level) buffer mappings on a set of benchmarks. In our analysis, the throughput is defined as the number of completed application iterations per processor clock cycle, where one application iteration corresponds to the completion of one execution of every actor. This is a common definition of throughout for SDF graphs, since SDF graphs typically execute iteratively across successive samples of the input signals that they are designed to process.

Our simulator for these experiments is a time-driven simulator developed in C. The processor-side simulator is developed at a high level of abstraction; only the estimated execution time of the actors is taken into account for the processor-side simulator. These estimates may be constant values or they may be drawn from probability distributions (e.g., to model the effects of infrequent events such as cache misses). Such a high-level simulation approach based on actor execution time estimates is useful in rapid prototyping for signal processing applications because it allows for accelerated processor-side simulation, and because execution time behavior in such applications has relatively high predictability. A global timer exists in the simulator, and each processor has its own local timer. The global timer is incremented by one on every cycle, while the local timer increases based on the execution times of actors. To determine whether a processor advances at a given point in the simulation, the simulator compares the global timer value with the processor's local timer value. DRAM-side simulation is carried out based on DRAMsim [16], which is a cycle-accurate, detailed, and highly-configurable C-based main memory system simulator.

We examine the synthetic and real benchmarks shown in Table 2. We use the TGFF algorithm to generate the synthetic benchmarks [17]. The benchmark application graphs are fairly complicated with 28-68 nodes, and the numbers of processors involved during scheduling ranges from 2 to 8.

Table 2. Benchmarks

Normal	(|V|, |E|)	# of proc.	# of IPC edges	Mem-intensive	# of proc.	# of IPC edges
fft1	(28, 32)	2	16	fft1_m	4	22
fft2	(28, 32)	3	20	fft2_m	6	23
qmf4	(14, 21)	2	10	qmf4_m	4	13
karp10	(21, 29)	3	16	karp10_m	6	20
tgff1	(20, 30)	3	16	tgff1_m	6	16
tgff2	(68, 119)	4	82	tgff2_m	8	86

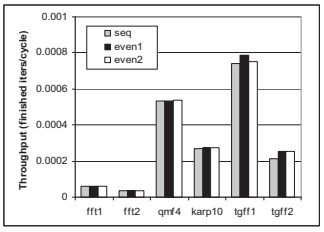

Figure 10(a). Throughput

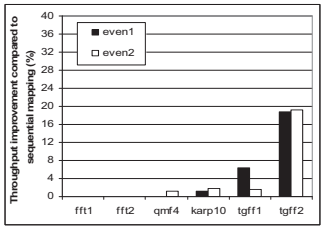

Figure 10(b). % improvement

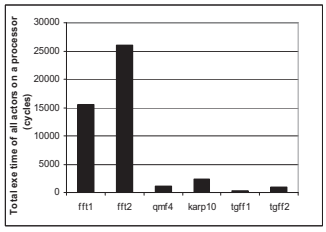

Figure 10(c). Total exe time of actors

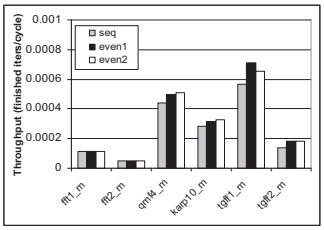

Figure 11(a). Throughput

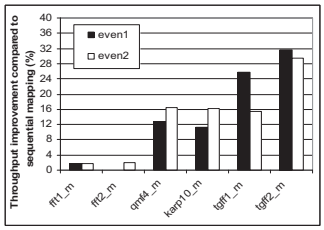

Figure 11(b). % improvement

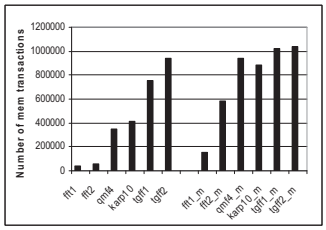

Figure 11(c). Number of mem trans

The examples fft1 and fft2 result from two representative schedules for Fast Fourier Transforms based on examples given in [18]; qmf4 is a 4 channel multi-resolution QMF filter bank for signal compression; and karp10 is a music synthesis application based on the Karplus Strong algorithm in 10 voices. In addition to experimenting with the normal benchmarks, we derive a set of memory-intensive benchmarks from the normal ones only by doubling the number of processors during the scheduling stage.

The DRAMsim parameters are set as follows: DRAM type = SDRAM, DRAM freq. = 100 MHz, # of ranks = adjusted, # of banks = 4, # of rows = 8192, # of columns = 512, transaction scheduling policy = first-come first-served, row buffer management policy = close page, address mapping policy = SDRAM base map, and refresh policy = all ranks and all banks at a time.

In Fig. 10(a), the measured throughput is shown for the three different buffer mapping policies on the six normal benchmarks. The percentage improvement of the even1 and even2 mappings compared to the sequential mapping is shown in Fig. 10(b). As shown in Fig. 10(b), the percentage improvement is very different from benchmark to benchmark; about 19% in tgff2 but 0% in fft1 and fft2. This is largely affected by the total execution time of all actors on a given processor. Fig. 10(c) shows the average total execution time of all actors on a processor. For example, even if we save 25000 cycles by applying the even1 mapping, this value is roughly translated to just one less iteration in fft2, but about 25 less iterations in tgff2. So the large execution times of fft1 and fft2 result in almost 0% improvement of the even mapping policies compared to the sequential mapping. The other interesting point is that the even2 throughput is smaller than even1 throughput in tgff1. This is due to the relatively low prediction accuracy of the sync_read check; the prediction accuracy in tgff1 is about 83%, but for the other benchmarks, the accuracy is almost 90%. As described in the previous section, the low prediction accuracy can cause useless read transactions to waste the memory bandwidth significantly.

To see the effect of applications' memory-intensity on the performance, we do the same simulations on the six memory-intensive benchmarks. Fig. 11(c) simply shows the increased memory intensity of the memory-intensive benchmarks compared to the corresponding normal benchmarks. The measured throughput is shown in Fig. 11(a) and (b). When examining these results, we see that first of all, the percentage improvement in the memory-intensive benchmarks is larger than that in the corresponding normal benchmarks except for fft2_m. In fft2_m, the even1 mapping policy does not show any improvement mainly because of its large total execution time. And in tgff1_m and tgff2_m, the even2 throughput is smaller than the even1 throughput due to the relatively high misprediction rate of the predictor.

Overall, the simulation results show that the proposed buffer mapping policy is very useful, especially in memory-intensive applications with relatively small total execution time of actors. Whether the even2 mapping is used or not should be determined depending on the sync_read prediction accuracy.

7 Conclusion and future work

In this paper, a high-performance buffer mapping policy was proposed for SDF-based DSP applications that are targeted to multiprocessor systems supporting the shared-memory programming model. The proposed policy exploits the bank and rank concurrency of contemporary DRAM main memory systems according to careful memory system modeling and parallelism analysis. A graph coloring technique was used to analyze IPC (inter-processor communication) edge-level parallelism and a 1-bit predictor was used to analyze comm/sync-level parallelism. In our experiments, we measured application throughput on both synthetic and real benchmarks. The simulation results show that the proposed buffer mapping policy is very useful especially in memory-intensive applications with relatively small total execution time of actors. Whether or not the even2 (comm/sync-level) mapping approach is used should be determined based on the sync_read prediction accuracy which can be estimated at compile time. The performance improvement of the proposed buffer mapping methodology is achieved in general at the cost of additional banks and bank under-utilization. Analytical analysis of IPC edge interference and a combined scheme to utilize both bank concurrency and page mode are useful directions for future work.

Acknowledgement

The authors thank Neal K. Bambha of the US Army Research Laboratory for providing his scheduling simulator. This research was supported in part by grant number 0325119 from the U.S. National Science Foundation.

References

[1] S. Sriram and S. S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc., 2000.

[2] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. In *Proceedings of the IEEE*, pages 1235-1245, Sep. 1987.

[3] M. Ade, R. Lauwereins, and J. A. Peperstraete. Buffer memory requirements in DSP applications. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 108-123, June 1994.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Converting graphical DSP programs into memory-constrained software prototypes. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 194-200, Chapel Hill, North Carolina, June 1995.

[5] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proceedings of the Global Telecommunications Conference*, Nov. 1989.

[6] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Optimizing synchronization in multiprocessor DSP systems. *IEEE Transactions on Signal Processing*, pages 1605-1618, June 1997.

[7] S. Rixner. Memory controller optimizations for web servers. *International Symposium on Microarchitecture*, pages 355-366, Dec. 2004

[8] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 128-138, 2000.

[9] R. Raghavan and J.P. Hayes. On randomly interleaved memories, *Proceedings of Supercomputing*, pages 49-58, Nov. 1990.

[10] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness, W. H. Freeman and Company, New York, 1999.

[11] Micron SDRAM 256Mb datasheet. http://download.micron.com/pdf/datasheets/dram/sdram/256MSDRAM.pdf.

[12] Elpida SDRAM 256Mb datasheet. http://www.elpida.com/pdfs/E0984E20.pdf.

[13] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 68-77, Oct. 1998.

[14] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis. A Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 39-48, Jan. 2000.

[15] D. Wang, Modern DRAM memory systems: performance analysis and a high performance, power-constrained DRAM-scheduling algorithm, PhD thesis, 2005.

[16] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A memory-system simulator. *SIGARCH Computer Architecture News*, pages 100-107, Sep. 2005.

[17] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free hardware/software codesign, In *Proceedings of the Sixth International Workshop on CODES/CASHE*, pages 97-101, Mar. 1998.

[18] C. L. McCreary, A. A. Kahn, et al. A comparison of heuristics for scheduling DAGs on multiprocessors. In *Proceedings of International Parallel Processing Symposium*, 1994.