Decoupled Access DRAM Architecture *

Alexander V. Veidenbaum
Dept. of Electrical Engineering and Computer Science
University of Illinois-Chicago
851 S. Morgan St, Chicago, IL 60607-7053
alexv@eecs.uic.edu

K. A. Gallivan
Computational Science and Engineering
and Department of Computer Science
Florida State University
Tallahassee, Florida
gallivan@cs.fsu.edu

Abstract

This paper discusses an approach to reducing memory latency in future systems. It focuses on systems where a single chip DRAM/processor will not be feasible even in 10 years, e.g. systems requiring a large memory and/or many CPU's. In such systems a solution needs to be found to DRAM latency and bandwidth as well as to inter-chip communication. Utilizing the projected advances in chip I/O bandwidth we propose to implement a decoupled access-execute processor where the access processor is placed in memory. A program is compiled to run as a computational process and several access processes with the latter executing in the DRAM processors. Instruction set extensions are discussed to support this paradigm. Using multi-level branch prediction the access processor stays ahead of the execute processor and keeps the latter supplied with data. The system reduces latency by moving address computation to memory and thus avoiding sending address to memory by the computational processor. This and the fetch-ahead capabilities of the access processor are combined with multiple DRAM "streaming" to improve performance. DRAM caching is assumed to be used to assist in this as well.

*This work was supported in part by the University of Illinois and National Science Foundation under Grant No. US NSF CCR-9796315.

1. Introduction

Memory access is a major problem in high-performance architectures. While processor speed has grown steadily with continuing advances in VLSI technology and attention to circuit design, DRAM speed has not kept. Both memory latency and bandwidth remain major problems in todays systems, uni- and multiprocessor alike. Recent developments, such as RAMBUS DRAM and its bus protocol [1], point to a solution for a problem of DRAM and bus bandwidth but more remains to be done to support higher ILP. The so-called intelligent or active memory RAM architectures [2, 3, 4, 5] have taken a different approach to the problem by avoiding off-chip access and exploiting on-chip access parallelism. Latency, however, remains a problem in both of these cases since they do not change the DRAM core organization.

Thus both uni- and multiprocessor systems continue to rely on a memory hierarchy to solve the memory access problem. However, the cost of cache misses is becoming prohibitive. RAMBUS and synchronous DRAM's utilize a form of on-chip caching and even more aggressive approaches have been proposed [3]. Latency hiding techniques, primarily prefetching [6, 7, 8], also have been utilized to help solve the problem.

Projected advances in VLSI and packaging technology are expected to make the problem much worse in the near future. The number of gates on a chip is projected to reach $20M$ in 10 years, while a DRAM can contain $2GBytes$ of data. This increase comes from

growth in die size, which increases interconnect distances on chip, and from feature size reduction. These lead to an increase in both on-chip interconnect length and propagation delays. The frequency of processor or logic ICs is expected to rise to 2GHz with up to 2×10^3 I/O's per chip. A chip of this size with a clock of 0.5ns may require 20 to 30 clocks to cross.

Given these semiconductor capabilities the approach combining a CPU with memory on the same chip seems to offer a hope for a system solution although the latency problem still remains. This solution, however, will not be sufficient for two important classes of future systems:

1. uni- or small multiprocessor (MP) single-chip systems with a large memory, e.g. high-end workstations and servers,

2. large MP systems.

A large memory here means more than 1 to 2GB on a single-chip DRAM which is expected in 10 years. But systems with this size memory are already needed and are built today, such as CAD workstations and database servers. Thus in the future a solution utilizing multiple DRAM ICs will be needed. Perhaps an even more important reason for a multi-chip system is that companies designing processors are likely to continue increasing processor power and thus transistor count and to make them as large separate chips which are proprietary and very expensive. They will have a very high ILP and require high memory bandwidth and low latency.

A solution for multi-chip systems assuming an integrated processor/memory IC has been proposed [10] relying on redundant use of hardware and cache coherence mechanism. This paper proposes a different multi-chip solution which can be applied to both uni- and multiprocessor systems. It assumes that development of super-CPUs will continue in the future in an attempt to exploit higher ILP [12, 16]. The question we ask is: given such a super-CPU chip of the year 2007 and multiple DRAM's, what can be done to help reduce memory latency and increase the bandwidth between CPU and memory?

The approach advocated here helps increase the DRAM bandwidth and reduce latency through decoupled access DRAM architecture and compilation. It is accomplished by placing a simple, small integer processor (or processors) on each DRAM to help generate enough memory requests to fully utilize the DRAM bandwidth. It is assumed that more external chip bandwidth to support this will be available within 10 years and that DRAM organization will change to provide more random-access internal bandwidth. Finally, caches are assumed to migrate to memory as well. The performance improvement is achieved by a fetch-ahead of the memory processor, by eliminating address transmission to memory, by building a cache hierarchy into the DRAM's, and by using multiple Crams simultaneously to send data to a single CPU. The resulting system is already a multiprocessor, albeit heterogeneous, and can be further extended to utilize multiple computational processors.

1.1. Decoupled Access-Execute Mechanism

Consider a uniprocessor system that consists of a processor, multiple DRAMs and interconnect. Traditionally, a processor in such a system performs loads and stores. The latter does not usually slow down the processor as stores can be cached, buffered, and retired later. Loads, on the other hand can stall the processor if data is needed and are a major cause of performance degradation. A problem with loads is that a processor needs to send the address to memory and then wait for memory to respond with data.

A Decoupled Access-Execute approach has been proposed and implemented [15, 13, 9, 14] to reduce the wait. The idea was to allow memory access instructions to be executed by a dedicated processor which communicates with a computational processor through a set of queues. The queues allow asynchrony in execution. Dynamically scheduled processors using Tomasulo's algorithm [17] implement a version of this approach with tagged queues. The details of organization and whether or not the queues are exposed to the user/compiler vary. A block diagram of such a decoupled processor is shown in Fig. 1.

In all cases the system is viewed as executing a single program which gets split by hardware into access and execute instructions. Another issue for both decoupled access processors and prefetching is conditional branch resolution. Since either processor can perform computation resolving a branch condition this information needs to be communicated to the other processor. Communication queues and $Branch\_on\_Queue$ instruction have been proposed to solve this problem ($E2A\ Br\ Q$ and $A2E\ Br\ Q$ in Fig. 1). More recently, branch prediction was utilized as well.

Another approach allowing some decoupling is prefetching, which attempts to predict the future addresses and initiate a memory read before the actual load is executed. Hardware, software, and integrated methods for doing it have been proposed. The hardware approach relies on a separate hardware engine to generate the addresses and a cache to store the arriving data. The CPU still executes all address computations

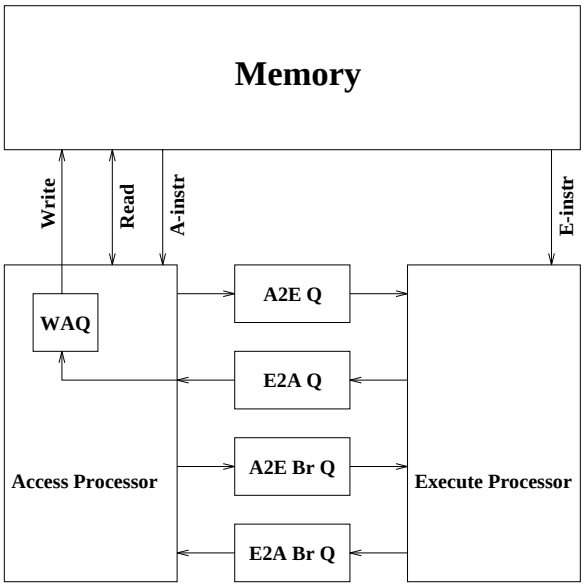

Figure 1. Decoupled Access System

and issues loads. Branch prediction remains a problem. So far prefetching has been typically confined to the processor or L1 cache. Moving it higher in the memory hierarchy is more difficult, although not impossible [19].

1.2. Decoupled Access Mechanism for DRAM's

The approach proposed here decouples uniprocessor computation and data access by allocating them to a computational processor (ComP) and one or more memory processors (MemP), respectively. It uses a compiler to generate code for a program in such a way that address computations and memory access are done by the memory processor. In some cases data computed by ComP may be required for address computations and will have to be sent to MemP. The MemP can be an integer processor. The computational processor is a complete, high-ILP processor. We assume that it does not execute memory loads. Rather it expects the data to be sent by the memory processor and fetches data from its queues. This is not the only possible implementation, but it is pursued here as the closest to the original ideas of decoupled execution. A block diagram of the decoupled DRAM system is shown in Fig. 2.

The MemP is capable of executing all integer instructions of the ComP and will execute address computations, control flow, etc. of a user program as well as the operating system. In addition, the instruction set is augmented with Send and Receive instructions for communication between the processors, including MemP-to-MemP communication. Other instruction set extensions are used as well and are discussed in more detail later in the paper. Examples illustrating the generated code are presented below.

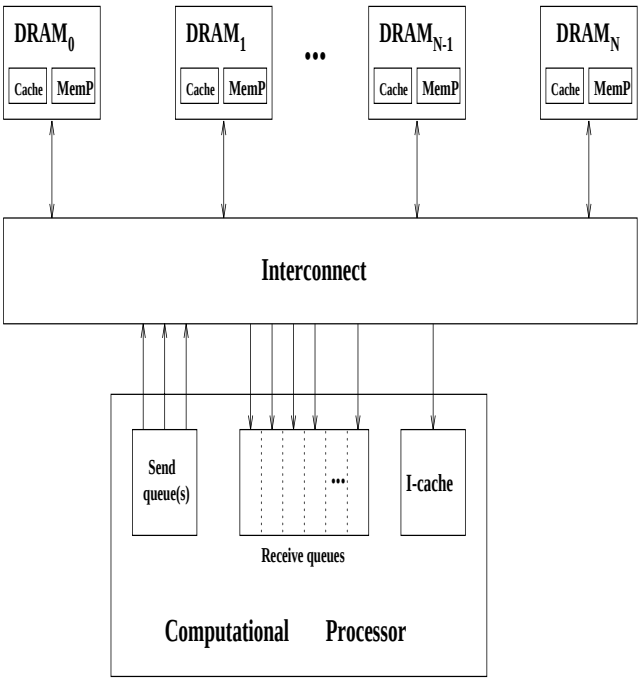

Figure 2. Decoupled Access DRAM System

The MemP attempts to run ahead of the ComP as is typical of decoupled access architecture. In many cases the MemP code can be generated in such a way that it can execute independently of the ComP and thus provide fetch-ahead capabilities. In other cases the code will require data computed in the ComP for conditional branches which will determine subsequent memory accesses. To solve the problem of conditional branching some code is replicated in both MemP and ComP, e.g. "static" loops. When data-dependent branching is present branch prediction is used. We propose to use multi-level branch prediction [20] to allow the MemP to advance without waiting for data from the ComP to resolve the branch condition. MemP can thus execute its program ahead of the ComP. Both may need to exchange branch information expected by the other. A roll-back mechanism is provided to undo misprediction. Data that is sent using a mispredicted branch is identified through tagging and not used by the receiver.

Multiple DRAM's can send multiple data items to the ComP every cycle and they can arrive out of order. The ComP contains queues to receive and reorder load data and to buffer outgoing stores. The ComP tags the stores and checks tags on incoming load data. The

MemP tags the loads (Sends) and receives and checks the store tags. The tag contains branch path information.

Internally, a ComP contains an instruction but no data cache. The use of multiple DRAM's is a major source of increased memory bandwidth and parallelism in accessing it. To make this a more powerful system, MemP'si n each DRAM are allowed to communicate directly in accessing the data. Thus multiple data streams can be automatically sent to the MemP. High chip I/O and interconnect bandwidth is assumed to be available for this. In order to send and receive multiple words to/from DRAM's each cycle, queues are partitioned. The decoupled nature of memory access and its exposure to the compiler are expected to allow a much higher memory request rate than a typical high-performance processor can achieve today. Each MemP can run a different program allowing even more flexibility.

To summarize, the decoupled DRAM approach has the following advantages:

1. It can eliminate the need for the main CPU to issue loads to memory.
2. It allows a significant lookahead in executing loads.
3. It simplifies the CPU address computation and partly eliminates it.
4. It allows address translation to be moved to memory.
5. It can allow the operating system execution in memory instead of ComP which may help the well-known poor memory hierarchy behavior of the OS.
6. It allows a large part of the memory hierarchy to be moved into the DRAM's.

1.3. Multiprocessing

So far only a uni-processor has been considered. However, the system using decoupled- access DRAM's is obviously a multiprocessor and supports Sends and Receives among processors. Thus one can say that by connecting multiple ComP's to multiple DRAM's and their MemP's one gets a multi-computer. Data layout optimization and programming to exploit it are going to be required for both uni- and multiprocessor systems. The major difference is direct support for shared memory which is carried from decoupled access DRAM uni- to multiprocessor system model. In fact, remote memory loads and stores also need to be supported making the DRAM and the ComP excellent MP building blocks.

Finally, the programming model for a uniprocessor using decoupled access DRAM's remains unchanged from a standard uniprocessor. But it requires a compiler to convert the program to a parallel decoupled access program. So to compile even for a uniprocessor requires MP concepts to be employed. This makes the unifying system model a multiprocessor.

Multiple decoupled access DRAM's are used in the uniprocessor case to provide sufficient bandwidth to the ComP. Use of multiple ComP's in uni- or multiprocessor organization will require more memory bandwidth. In both cases, to achieve this may require multiple MemP's to be placd in each DRAM. Other ways of increasing the DRAM bandwidth may need to be utilized as well. Finally, the latency may still remain too high, if sufficient lookahead cannot be achieved. These issues are discussed in the next section.

1.4. High-bandwidth, low latency DRAM

The use of a memory processor MemP leads to latency reduction due to shortened address transmission time and fetch-ahead. To support it, a DRAM needs to provide more internal as well as off-chip bandwidth as well as to reduce the latency of accessing the memory. The projected increase in the number of package I/Os and their operating frequency will provide the off-chip bandwidth. This has already been demonstrated by RAMBUS DRAM [1] ad will advance even further in spite of many technical hurdles, such as power dissipation, noise immunity, etc. The MemP will help better utilize a large number of DRAM I/O's.

Internally to the DRAM plenty of bandwidth potential exists but modifications are required to extract it. For instance, RDRAM design required a wider column pitch to increase the access bandwidth. Some existing DRAM's already utilize a row of sense amplifiers as a latch for temporary data storage. Much discussion has taken place about the use of these latches to form a cache. These latches are neither large enough or sufficiently fast to act as a cache for a 1 to 2GB DRAM. Current high-performance workstations use 2-4MB of cache for 256MB of memory.

Mitsubishi CDRAM used a different approach and implemented a 16Kbit cache in a 4Mbit DRAM. This required very little extra real estate, about 7%, and provided a much better cache albeit still too small for a large memory system.

In this paper, a traditional multi-bank DRAM organization is assumed, each bank possibly operating completely independently to increase parallelism. By allocating more real estate, each bank may be provided with a large enough cache to satisfy a large percent-

age of accesses. In addition, a large memory cache is assumed at DRAM I/Os. Given a high degree of on-chip DRAM interleaving via multiple independent banks and caching, the MemP needs to be able to utilize it. This may require the use of several MemPs.

We have shown in the past that memory caches are very effective in multiprocessors [21] due to the lack of the coherence problem. The decoupled architecture makes it even more advantageous to move caches to memory and rely on them to further reduce DRAM latency. An important question is the organization of the memory cache. We see the role of the memory cache in hiding the increased on-chip interconnect latency. Thus some form of geometrically distributed cache hierarchy may be necessary to avoid long-distance propagation on chip. It will consist of a large L1 cache for MemP, which is also the main memory cache, followed by smaller, sub-array caches distributed over the DRAM chip. Sense amplifier cache can still be used as the last level before memory.

2. Details of the organization

To support decoupled execution the following architectural issues have to be addressed and are discussed in this section:

1. Branch prediction
2. Remote memory access
3. Inter-processor Send and Receive instructions
4. Other instruction set extensions
5. A tagging mechanism for load and store data
6. Flow control and synchronization of data interchange

It is assumed that the combined multiple DRAM storage capacity forms the total shared physical address space of the system. Some storage is reserved as private in each DRAM. Let us assume for simplicity that each DRAM contains the complete user program as well as the operating system. These are stored in the private space of each DRAM. The number of DRAM's in the system and each DRAMs id are available to its MemP. The address space is block interleaved across DRAM's, with block size smaller than a page. This allows a DRAM id number to be extracted from a virtual address without translation. All translation is assumed to be performed in memory and each DRAM is responsible for translation of all its addresses.

Loads and stores are processed asymmetrically by the ComP. The load data is expected to come from architecturally visible queues without address computation and memory access. Store addresses are computed explicitly and stores buffered in the send queue. The reason for treating stores this way is to obtain the address and from it the DRAM id used to send data. To assist decoupled processing by separate MemP and ComP both loads and stores are tagged with a value number. For example, the induction variable can be used as the value number or the value of a compiler-specified variable. The value number also includes N bits of branch history to precisely identify data.

Operand queues
Each processor contains a number of queues for storing/sending and loading/receiving data. The two types of queues are organized differently. A load queue is assumed to be organized as a small, associative cache with a tag to assist in operand reordering. It is assumed there is a load queue for each register r_i accessible by LDr_i. A store queue is logically a single queue.

Data Communication
MemP and ComP can issue tagged Send and Recv instructions to communicate. The tag is appended on a Send and checked on a Recv. Send is non-blocking while Recv is a blocking operation. A Send/Recv pair uses the same register and thus the same queue. Send can be cancelled if it used an invalid register. Recv searches the input queue using the expected value number. The instruction formats are as follows:

Send r_i, dest
Recv r_i

Remote Memory Access
MemP and ComP can issue tagged remote load (LD) and store (ST) instructions. These remote LD/ST instructions are routed to the requested DRAM, translated there, and memory is accessed. The requested DRAM id comes from an address in register r_j. The remote memory request specifies the node and register to which the load data is returned. The destination node "dest" is an extra specifier in the instruction:

LD_r r_i, (r_j), dest

A load request also carries the requesters tag. As mentioned above, the ComP program does not utilize loads directly. Instead these are used for MemP remote memory access, but by specifying a ComP as a destination, indirect addressing can be sped up. LD_r is equivalent to a "standard" load followed by a Send.

Local Memory Access
In addition to standard LD/ST instructions a "local-only" version is provided,

$LD\_l\ r_i$, offset(r_j)

This instruction computes the address and only issues a load or a store if the address is local to this node. Otherwise the load is cancelled and the register r_i is marked invalid. Any instruction using an invalid input register is cancelled as well and its destination register marked invalid. Any of the load or Recv instructions described above can make a register valid.

Conditional Branches
The MemP may require a data value from ComP to resolve a conditional branch. In such a case a compiler inserts a new instruction, BRanch_on_Prediction, in the MemP code. The ComP sends n bits of its branch history as they are accumulated. MemP also keeps track of its branch history and compares it with the history bits received from ComP. In case of a discrepancy the MemP rolls back the execution, updates its predictor and history register, and re-starts execution. Any data sent by the MemP during execution of the mispredicted path is detected by the ComP when it checks the Recv queue tag.

Standard conditional branches are also executed by both processors. It is assumed that ComP does not execute Branch_on_Prediction but rather receives the data and does its own testing.

Synchronization
Every so often it may be necessary to synchronize all processors to know that they all have a consistent state with respect to either data or conditional statements. A barrier instruction *BAR* is provided for this purpose. Memory consistency model relies on the barrier and can be either strong or relaxed. BAR can be used to trigger synchronization and backtracking in case of branch misprediction.

Data Dependences
With decoupled execution one has to check RAW and WAR dependencies. WAW will be guaranteed by hardware. Normally it is not hard to do, but without load addresses it is more difficult. Value tags will be used instead and send queue checked on reads. The checking will continue all the way to MemP and its queues.

Caches
MemP uses standard memory hierarchy but for ComP tagged decoupled access makes the use of caches more difficult. ComP will have an instruction cache. The tagged queues act a bit like a data cache. If a true data cache is desired it can be tagged with a value number and branch history rather than address. The address must be present in the cache to implement a write-back policy.

3. Compilation

The instruction set extensions and their semantics for decoupled processing define the compilation approach. In addition, a programming model needs to be defined, especially since multiple processors now cooperate in executing the single program. We discuss it below assuming a "uniprocessor" case, e.g. one ComP and multiple MemP's. Partitioning the program into access and execute programs for this architecture is shown through examples in the next section.

It is assumed that any computation involving variable referencing is pulled into a MemP program. The ComP program loads are replaced with queue access while stores are left as they were. Most simple control structures such as loops with index variables are duplicated in both places and necessary data is sent by a MemP to ComP. We are not concerned with this redundant integer computation since integer processing is relatively inexpensive. The ComP must compute the address of stores to determine to which DRAM the data must be sent. A MemP cmputes the address for loads that are usually independent of data computations, using data in memory. But occasionally, a ComP and a MemP may have to exchange data.

For simplicity, the examples in this paper assume that both MemP and ComP programs are extracted from the same "original program" and keep same names, including register names, identical in both. While this places some restrictions on code generation and optimization for each type of processor, it is a valid initial assumption. Each MemP is assumed to execute a complete memory access program but access only local data. This is similar to what is known in the parallel community as the SPMD model of computation. In this case it is further assisted by the fact that a complete program exists in each node. Thus each MemP has access to symbol table information without a need for communication. Each MemP executes the program but only accesses data stored in its memory. This it can figure out from the virtual address. Data stored locally is fetched and sent to the ComP with an appropriate tag. If a MemP finds it does not have the data locally it skips the computation. In some cases complicated indirection is used to address data needed by ComP. This can mean that data accessed locally is used to compute an address of data needed by the

processor, e.g. A(B(i)). This requires MemP to MemP communication.

4. Examples

In this section, we consider two simple loops. They are hand-compiled using the instructions described earlier using the SPMD programming model. Both MemP and ComP codes are shown. A comparison with a "standard" uniprocessor execution is made and qualitative performance differences discussed.

4.1. Example 1: Vector Loop

Let us start with a vector code without conditional statements. Consider the SAXPY loop below.

```
do 100 i=1,N
       Y(i) = Y(i) + a * X(i)
100 continue
```

The simplified assembly code followed by the decoupled

```
      LD      r1, addr_X    ; address of X
      LD      r2, addr_Y    ; address of Y
      LD      f0, a         ; a
      LD      r3, N         ; loop ctr=N
 L:   LD      f2, 0(r1)     ; X(i)
      FPMUL   f4, f2, f0
      LD      f6, 0(r2)     ; Y(i)
      FPADD   f6, f6, f4
      ST      f6, 0(r2)     ; Y(i)
      ADD     r1, r1, #8
      ADD     r2, r2, #8
      SUB     r3, r3, #1    ; loop count
      BEQZ    r3, L
```

code for the MemP and the ComP are shown. It assumes that each DRAM contains the program and knows the number of DRAM'ss in the system and its DRAM id. Each MemP only sends to ComP data stored in its DRAM. Global value numbering is used, however. The example assumes that stores and their address computation are performed by ComP. The compiler generates MemP code which specifies the register in ComP into which the data must be placed. In this case "dest" field is set to ComP.

What is the difference in performance between the decoupled and a standard processor on this code? The decoupled system can theoretically stream N words per cycle to the ComP, where N is the number of DRAM's. ComP can accept them given multiple ports/queues and issue multiple computational instructions per cycle. This is from memory and independent of cache

MemP		ComP	
Sync		Sync	
LD	r1, addr_X		
LD	r2, addr_Y		
Send	r2, dest	Recv	r2
LD	f0, a		
LD	r3, N		
Send	f0, dest	Recv	f0
Send	r3, dest	Recv	r3
L:			
LD	f2, 0(r1)		
Send	f2, dest	L:	
		Recv	f2
LD	f6, 0(r2)	FPMUL	f4, f2, f0
Send	f6, dest	Recv	f6,
		FPADD	f6, f6, f4
ADD	r1, r1, #8	ST	f6, 0(r2)
ADD	r2, r2, #8	ADD	r2, r2, #8
SUB	r3, r3, #1	SUB	r3, r3, #1
BEQZ	r3, L	BEQZ	r3, L

performance. It would be difficult for the L1 cache or the hierarchy to supply so many words per cycle, requiring multiporting and a large number of simultaneously outstanding requests.

The above code can run in one or multiple MemP without change if SPMD mode of execution is used. Or it can be re-written as a FORALL loop and "tuned" to execute N/P iterations per processor computing addresses of only those items residing in a given memory. In this case, there is no communication required between different MemP. This is not the case in the following example.

4.2. Sparse matrix times dense vector

This is a simplified sparse matrix times a dense vector kernel. It assumes consecutive storage for the rows of the matrix. It is used to illustrate the case when MemP to MemP communication can be used to improve performance. The multiply code is shown below with its storage declarations:

```
sm(1:non_zero)   -  "compressed row" storage
dcv(1:colmns)     -  dense column vector
res(1:rows)      -  result column vector
ptr(1:rows+1)  -  ptrs to sm's row starts
indx(1:non_zero) -  column numbers of sm

        do 200 i = 1, rows
           kl = ptr(i)
           ku = ptr(i+1) -1
           t  = res(i)
```

```
          do 100 k = kl, ku
                 t = t + sm(k) * dcv(indx(k))
100       continue
          res(i) = t
200     continue
```

The simplified assembly for the whole loop nest is shown in Appendix A. Shown below is the assembly for just the inner loop and the subsequent store. The

```
L2:                         ; inner loop
LD      r15, 0(r9)          ; indx(k)
LD      f2, 0(r10)          ; sm(k)
                            ; dcv addr
SUB     r15, r15, #1        ; indx(k) -1
ASL     r15, r15, #3        ; 8*(indx(k)-1)
ADD     r15, r15, r11       ; addr dcv()
LD      f3, 0(r15)          ; dcv((indx(k))
FPMUL   f4, f2, f3
FPADD   f1, f1, f4
ADD     r10, r10, #8        ; incr sm addr
ADD     r9, r9, #4          ; incr indx addr
SUB     r5, r5, #1          ; loop count k
BEQZ    r5, L2
        ...
ST      f1, 0(r2)           ; res(i)
```

inner loop of the program converted to decoupled execution is shown next. Again, simple SPMD execution is assumed. In this case only local loads are executed by one MemP for a given iteration and become no-ops in all others. Finally, the node computing the address and finding it to be local executes a remote load and specifies ComP as the receiver.

To compare performance with a standard processor note that here all MemP's quickly get through the iteration and move on to the next one except for the two per iteration that actually access first their local and then remote memory. However, with SPMD model it is hard to expect more parallelism in memory access. This may not be sufficient to keep the processor busy.

The inner loop can be re-written as a FORALL and "tuned" to execute N/P iterations per processor. In this case, however, it is hard to compute the addresses of only those items residing in a given memory. MemP to MemP communication would be required but will result in more parallelism and better memory utilization. The resulting code is shown next and has the potential to exploit all the available memory bandwidth. For simplicity, the required changes in index variable and address computations are not shown.

As a result, now all participating MemP's are generating memory requests at the same time and can "stream" data to ComP. However, remote loads are

MemP		**ComP**	
L2:		L2:	
LD	f2, 0(r10)		
Send	f2, ComP	Recv	f2
LD_l	r15, 0(r9)		
SUB	r15, r15, #1		
ASL	r15, r15, #3		
ADD	r15, r15, r11		
LD_r	f3, 0(r15), ComP		
		Recv	f3
		FPMUL	f4, f2, f3
		FPADD	f1, f1, f4
ADD	r10, r10, #8	SUB	r5, r5, #1
ADD	r9, r9, #4		
SUB	r5, r5, #1		
BEQZ	r5, L2	BEQZ	r5, L2
		...	
		ST	f1, 0(r2)
		...	

used for this. The main advantage of a remote load is its ability to send results back to a specified processor without coming back to the requesting processor first. This works well for two out of three loads. The third one requires indirection. An approach combining FORALL computation with identifying local data is what is needed for best performance.

5. Conclusions

This paper presented a possible approach to increasing memory bandwidth and reducing latency by using a decoupled access-execute paradigm. The major innovation comes from moving the access processor to memory and compiling programs to execute as separate but communicating processes in memory and the computation processor. The need for such a system comes from two sources: systems requiring multiple DRAM's and Super-CPU's with extremely high ILP that cannot be integrated with memory on one chip.

The system we propose is an evolutionary step from separate processor-memory architectures towards an integrated processor-memory system. It takes the idea of decoupled access-execute architectures and partitions the system to place a simple access processor and cache in memory. Eventually it may be possible to put the ComP in memory as well allowing the approach proposed by Goodman et al for multi-chip systems. The broadcast involved in this solution is its main drawback and perhaps hybrid approach is possible. It is argued that given sufficient memory and communication bandwidth the system described here

MemP		ComP	
L2:		L2:	
LD_r	f2, (r10), ComP	Recv	f2
LD_r	r15, (r9), myid		
SUB	r15, r15, #1		
ASL	r15, r15, #3		
ADD	r15, r15, r11		
LD_r	f3, 0(r15), ComP	Recv	f3
		FPMUL	f4, f2, f3
		FPADD	f1, f1, f4
ADD	r10, r10, #8	SUB	r5, r5, #1
ADD	r9, r9, #4		
SUB	r5, r5, #1		
BEQZ	r5, L2	BEQZ	r5, L2
		...	
		ST	f1, 0(r2)
		...	

can reduce latency by aggressively streaming data from multiple memories to a computational processor.

Another way to reduce latency is via DRAM caching which was briefly discussed. Overall, the idea of memory cache is very appealing. The qualitative discussion of performance potential showed this a promising approach.

Finally, one can observe similarity between prefetching and decoupled access. Prefetching engines, even the explicitly programmable ones, are simpler than a decoupled processor. They can also be added to a standard system without the major change in the processor architecture required by the decoupled system described here. They can also peacefully coexist with the CPU caches. Thus a intermediate solution may be possible which adds an even simpler MemP to each DRAM but leaves the computational processor and its program largely the same. We are currently exploring this possibility.

References

[1] RAMBUS DRAM Data sheets and application notes, *RAMBUS Inc,* 1997

[2] David Patterson et al, A Case for Intelligent RAM: IRAM *IEEE Micro,* April 1997

[3] Saulsbury, A., Pong, F., and Nowatzk, A. Missing the Memory Wall: the Case for processor/Memory Integration *Proc. 23rd Int. Symp. on Computer Architecture,* 1996.

[4] Peter M. Kogge et al, Combined DRAM and logic chip for massively parallel systems, *Proc. 16th Conference on Advanced Research in VLSI,* pp.4-16, 1995

[5] Murakami, K, Shirakawa, S, amd Miyajima, H., Parallel Processing Chip with 256Mb DRAM and Quad Processor, *Proc. IEEE Int. Solid-State Circuits Conference,* 1997

[6] Baer, J.-L. and Chen, T.-F., An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing*, pages 176–186. IEEE, November 1991.

[7] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pp. 354-368, 1990.

[8] Mowry, T.C., Design and evaluation of a compiler algorithm for prefetching. In *PhD Thesis, Computer Science Dept., STanford University,*, 1994.

[9] Goodman, J., Hsieh, J., Liou, K., Plezkun, A., Schecteur, P., Young, H. PIPE: A VLSI Decoupled Architecture. *Proc. 12 Int. Symp. on Computer Architecture,* June 1985.

[10] Burger, D., Kaxiras, S. and Goodman, J.R., DataScalar Architectures *Proc. 24 Int. Symp. on Computer Architecture,* 1997.

[11] Palacharla, S. and Smith, J.E., Decoupling Integer Execution in Superscalar Processors, *Proc. IEEE Intnl. Symposium on Microarchitecture,* pp.285-290, 1995

[12] Smith, J.E., et al. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching *Proc. IEEE Int. Symp. on Microarchitecture,* 1997.

[13] Smith, J.E., et al. The ZS-1 Central Processor. *Proc. 2 Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 1987, Palo Alto, CA.

[14] Wulf, Wm. A, An Evaluation of the WM Architecture *Proc. Int. Symp. on Computer Architecture,* May 1992, Gold Coast, Australia.

[15] Pleszkun, A.R. and Davidson, E.S., Structured Memory Access Architecture, *Proc. Intnl. Conference on Parallel Processing,* pp.461-471, 1983

[16] The Intel Merced processor, *Microprocessor Report,* 1997

[17] Tomasulo, R.M., An Efficient Algorithm for Exploiting Multiple Arithmeitc Units, *IBM Journal of Research and Development,* pp. 25-33, Jan.Jan. 1967

[18] Mitsubishi CD-ROM Data sheets and application notes, *Mitsubishi Electronics,* 1995

[19] Sunil Kim and Alex Veidenbaum, Stride-directed Prefetching for Secondary Caches, *Proc. International Conference on Parallel Processing, pp. 314-321,* Aug. 1997

[20] Yeh, T-Y, Marr, D.T., and Patt, Y.N., Increasing Instruction Fetch Rate Using Multiple Branch Prediction and a Brach Address Cache, *Proc. International Conference on Supercomputing,* pp. 67-76, 1993

[21] Chen, Y.-C. and Veidenbaum, A.V., Performance Evaluation of Memory Caches in Multiprocessors, Proc. 1993 Int'l Conference on Parallel Processing, Aug. 1993.

A. Sparse Matrix Multiply Assembly

Shown below is a simplified assembly program for a sparse matrix times a dense vector code segment discussed in the paper.

```
LD      r1, addr_ptr     ; addr of ptr
LD      r2, addr_res     ; addr of res
LD      r3, rows         ; loop count
LD      r6, addr_indx    ; addr of indx
LD      r11, addr_dcv    ; addr of dcv
LD      r8, addr_sm      ; addr of sm
L1:     .....            ; outer loop label
LD      f1, 0(r2)        ; load res(i)x
LD      r4, 0(r1)        ; k=kl=ptr(i)
LD      r5, 4(r1)        ; ku=ptr(i+1)
SUB     r5, r5, #1       ; ku=ptr(i+1)-1
ADD     r7, r4, #-1      ; k-1
SUB     r5, r5, r7       ; ku-kl+1
                         ; indx addr
ASL     r7, r7, #2       ; 4*(k-1)
ADD     r9, r7, r6       ; indx(k) addr
                         ; addr of sm
ADD     r10, r7, r7      ; 8*(k-1)
ADD     r10, r10, r8     ; addr of sm(k)
L2:     .....            ; inner loop label
LD      r15, 0(r9)       ; indx(k)
LD      f2, 0(r10)       ; sm(k)
                         ; dcv addr
SUB     r15, r15, #1     ; indx(k) -1
ASL     r15, r15, #3     ; 8*(indx(k)-1)
ADD     r15, r15, r11    ; addr dcv()
LD      f3, 0(r15)       ; dcv((indx(k))
FPMUL   f4, f2, f3
FPADD   f1, f1, f4
ADD     r10, r10, #8     ; incr sm addr
ADD     r9, r9, #4       ; incr indx addr
SUB     r5, r5, #1       ; loop count k
BEQZ    r5, L2
ST      f1, 0(r2)        ; res(i)
ADD     r2, r2, #8       ; incr res addr
ADD     r1, r1, #4       ; incr ptr addr
SUB     r3, r3, #1       ; loop count i
BEQZ    r3, L1
```