PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

DATA DIFFUSION MACHINE
–A SCALABLE SHARED VIRTUAL MEMORY MULTIPROCESSOR

David H. D. Warren
Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Seif Haridi
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

ABSTRACT

The Data Diffusion Machine (DDM) is a scalable shared virtual memory multiprocessor where the location of a datum in the machine is completely decoupled from its virtual address. In particular, there is no distinguished home location where a datum must normally reside. Instead data migrates automatically to where it is needed, reducing access times and traffic.

The hardware organisation consists of a hierarchy of buses and data controllers linking an arbitrary number of processors each having a large set-associative memory. Each data controller has a set-associative directory containing status bits for data under its control. The controller supports remote data access by "snooping" on the buses above it and below it. The data access protocol it uses provides for the automatic migration, duplication and replacement of data while maintaining data coherency.

The machine is scalable in that there may be any number of levels in the hierarchy. Only a few levels are necessary in practice for a very large number of processors. Most memory requests are satisfied locally. Requests requiring remote access generally cause only a limited amount of traffic over a limited part of the machine, and are satisfied within a small time that is logarithmic in the number of processors. Although designed particularly to provide good support for the parallel execution of logic programs, the architecture is very general in that it does not assume any particular processor, language or class of application.

1 INTRODUCTION

Inspired partly by the ambitious goal of Japan's Fifth Generation project to develop a machine capable of one billion logical inferences per second, we and our colleagues have been investigating models for transparently exploiting parallelism in logic programs. In particular, we have developed two models: the SRI model [7], which has recently been implemented in the prototype system Aurora [5], and a more recent generalisation of it called the Andorra model [10,2].

In the SRI model, or-parallelism is exploited through a coordinated exploration of a search tree by a group of processing agents called workers. Each worker is equipped with a binding array to give it fast access to the variable bindings relevant to its current position in the tree. The Andorra model generalises the SRI model by allowing several workers to work in and-parallel on determinate goals arising on one branch of the search tree, while continuing to support or-parallelism through different teams of workers exploring different branches. In the SRI model, the parallel tasks are relatively independent and involve little interaction between workers. In contrast, in the Andorra model, workers working on the same branch are creating a shared set of variable bindings and will typically have to interact much more closely.

The SRI, Andorra, and other models (e.g. Pepsys [8]) share a common general approach to parallel computation, whose scope is by no means confined to logic programs. In this approach, the state of computation is viewed as a single large data structure. A number of processing agents, or workers, work in parallel on different parts of the data structure. The aim is that each worker should work as fast as if it alone were performing the computation. Thus interaction between workers should be minimised, and what each worker does should approximate a standard sequential computation. From this viewpoint, parallel computation is little different from any other parallel activity where there is a group of agents operating in parallel on a shared structure, for example a team of workmen building a house.

How can hardware best support this view of parallel computation? Abstractly, there is a need for shared data to be accessed freely and uniformly by multiple processes running on multiple processors. Each data item needs some unique name, which is nothing more than a virtual address. This leads us to propose that the machine, whatever its precise hardware organisation, should be a **shared virtual memory architecture**, that is one which allows software to access all data uniformly via a global virtual address space.

The most obvious way to implement the abstract requirement of shared virtual memory is through shared physical memory. Indeed, the shared virtual memory architectures that exist today are typically shared (physical) memory machines, e.g. Sequent and Encore. It is on these machines that the prototypes of our execution models currently run. However it is important to realise that shared virtual memory architecture need not necessarily be implemented in this way. The purpose of this paper is to describe a novel shared virtual memory architecture, the **data diffusion machine** (DDM), which is not based on shared physical memory. Rather, its hardware organisation is in many ways closer to a message-passing machine.

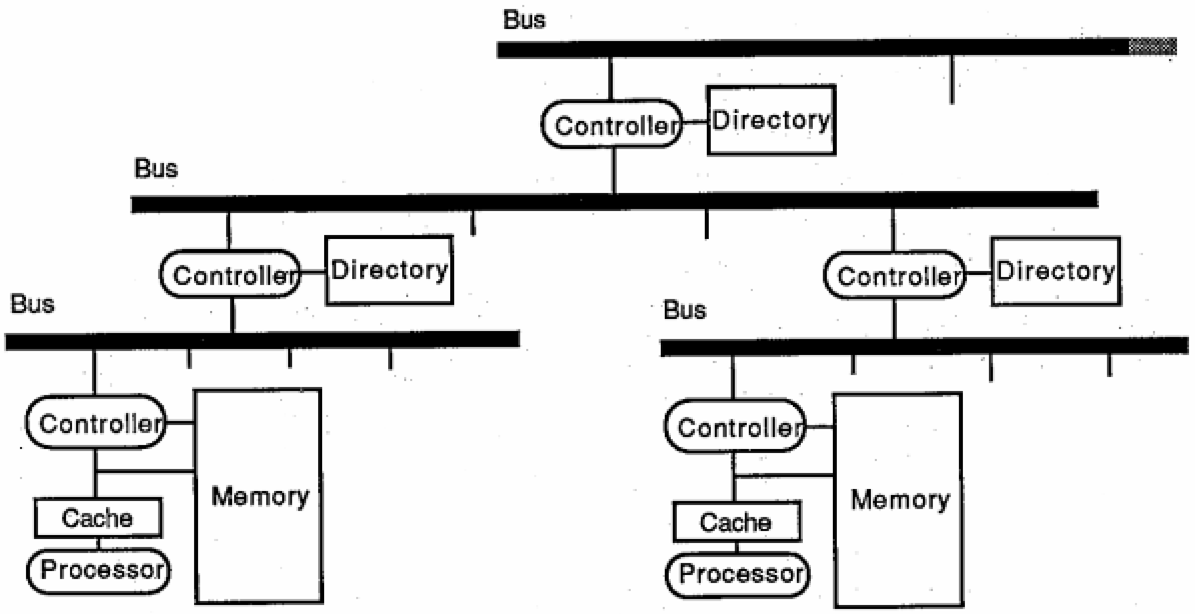
Message passing machines and shared memory machines are the two main classes of parallel (MIMD) computer, and are generally considered to be quite distinct. Message passing machines typically have many processors with large private memories, linked together by a communications network. Shared memory machines typically have only a limited number of processors with small private memories or caches, connected by a common bus to a large, physically shared, memory. Message passing machines usually require software to view memory access and communication with other processors as quite separate mechanisms. (Software is often therefore driven to simulate a form of shared virtual memory by translating references to remote objects into appropriate messages). Shared memory machines, on the other hand, usually support shared virtual memory directly, thereby allowing software to achieve communication implicitly through memory access, but require some locking mechanisms to support this. Message passing machines are generally **scalable** to arbitrary numbers of processors, whereas in shared memory machines the shared bus and memory is a bottleneck, placing a limit on the number of processors that can be attached. However, message passing machines place a much heavier burden on software to partition the computation effectively, and so the scalability of the hardware is only useful in so far as the software can keep communication to a minimum.

The DDM is like a message passing machine in that memory is distributed and the machine is scalable to an arbitrary number of processors. The DDM is like a shared physical memory machine in that it supports a shared address space by connecting processors via shared buses (using a "data coherency protocol"). The key idea behind the DDM, which distinguishes it from both message passing machines and shared memory machines, is that the location of a data item in the machine is completely decoupled from its virtual address.

The design of the DDM is based on the following considerations. Where a piece of data resides is not really relevant to the software. Ideally, the physical location of data should be completely transparent to software, and placement should be controlled automatically by hardware. Thus virtual addresses should be mapped into physically addresses in a totally flexible manner. The mapping should be dynamic, allowing data to migrate to where it is most needed. It may be desirable to have multiple copies of a particular data item, but they will all share the same virtual address. To summarise, from a software point of view there will be of a number of processes sharing data that is arranged logically in a single virtual address space; from a hardware point of view, processes will be mapped into processors and virtual addresses into physical addresses in such a way that most of a processor's memory accesses can be satisfied by its local memory. In other words, the data structure that the software sees will distribute itself automatically over the machine in such a way as to reduce data access times and minimise data traffic.

The DDM was motivated by our work on logic programming execution models and represents our ideas on how these models can best be supported by hardware. However the design is very general in that it does not assume any particular kind of processor, language or application. We feel this is very important if the machine is to gain practical acceptance, and is an important factor in the commercial success of machines such as the Se-

Figure 1: Data Diffusion Machine



quent. It should be noted that software designed for conventional shared memory machines can run without change on a DDM.

The remainder of the paper is organised as follows. In the first section we describe the main features of the hardware organisation and summarise the machine's main principles of operation. The next section gives the details of the data access protocols which control the distribution of data. Next, we analyse the main performance characteristics of the machine and compare it with other multiprocessors. In the following section, we discuss various other issues necessary for a complete machine. We conclude with a summary of the main novel features of the design.

2 HARDWARE ORGANISATION

The machine is hierarchical (see Figure 1). At the tips of the hierarchy are processors each with a large local memory (possibly accessed via a conventional cache). The memory contains an image of some part of the global virtual address space. The memory is set-associative, and is organised like a (very large) cache, but it should be emphasised that this is the sole form of main memory in the machine. The memory is connected via a memory controller to a local bus. The local bus connects a cluster of similar configurations of processor, cache, memory and controller. The local bus may itself be connected via a controller to a higher bus, and so on up the hierarchy. The higher level controllers each have access to a directory of status information, and are termed directory controllers. The directory is set-associative, and has space for status bits for all the data items in the memories below.

The function of a controller is to mediate between the bus above it and the subsystem below it. Its behaviour is a generalisation of the "snooping" caches in single-bus shared memory processors. It allows memory requests to be handled as locally as possible, but where a request cannot be handled locally, it is responsible for transmitting that request upward or downward to enable it to be satisfied. The controller has access to a directory which tells it which part of the virtual address space is mapped into the memory of the subsystem below it, and whether any of those virtual addresses are also mapped into memory outside the subsystem. Thus for any virtual address, the controller can answer the questions "Is this datum below me?" and "Does this datum occur elsewhere (not below me)?".

The behaviour of the controllers is such that a memory request will not be transmitted outside a subsystem if (1) it is a read of a local datum or (2) it is a write to an unshared local datum. In particular, this means that if a processor tries to read a datum in its local memory or write an unshared datum in its local memory, no external communication is required. Normally, this will cover the vast majority of memory references.

If a subsystem tries to read a non-local datum, the read request will be transmitted as far as is necessary to retrieve a copy of the datum, and the datum will be marked as shared where necessary. If a subsystem tries to write a shared datum, requests will be propagated to erase all other copies of the datum, and the datum will then be marked as unshared. It would of course be possible to update other copies rather than erase them. However, erasing on write has the advantage of helping to concentrate data where it is being actively used, and avoids the overheads of continually updating "stale" data. If a memory becomes full, data items that are shared elsewhere can be discarded. The machine will select items which are least recently used. If there is no such item, an exclusive item that is least recently used will be moved elsewhere. This is another means by which data tends to reside only where it is being actively used.

The following points should be noted. The data that a processor creates itself will automatically reside in its own memory and will not be copied anywhere else unless another processor requires it. A processor is likely to spend most of its time accessing such data. A processor is not obliged to repeatedly access a datum from a remote memory, if the data is initially remote. Instead, remote data tends to migrate to where it is being actively used, and is not tied to some fixed "home" location.

3 DATA ACCESS PROTOCOLS

In describing in more detail the data access protocols, the following assumptions are made about the DDM.

At the lowest level, the system consists of a number of processors. Each processor is connected via a memory controller to a memory and a bus. At higher levels, there is a directory controller for each subsystem. Each bus is synchronous and has its own clock. There is an arbiter on each bus to select one request. Each virtual address in a memory is in one of the following states:

invalid (I) The datum is invalid (does not exist) inside.

exclusive (E) The datum exists inside but not outside.

shared (S) The datum exists inside and also outside.

writing (W) The datum is being written inside and is waiting for invalidation outside.

reading (R) The datum is being read inside and is waiting for a response from outside.

The states R and W are transient and would not be needed in a single bus multiprocessor.

A memory is N-way set-associative, and stores data items plus their associated status bits. A directory is also N-way set-associative but with a size equal to the total number of entries in the memories below it. Directories have space to store only status bits. If the directories were perfectly associative, there would always be space in the directory for the items below. However, because the directories are only set-associative, there can be situations where a requested data item that has a space at a lower subsystem cannot be guaranteed space for its status bits at a higher level directory. This issue is discussed further in the section concerning the data replacement strategy.

Each datum in a directory is in one of the following states according to its status within the subsystem below:

invalid (I) The datum is invalid (does not exist) inside.

exclusive (E) The datum exists inside but not outside.

shared (S) The datum exists inside and also outside.

writing (W) The datum is being written inside and is waiting for invalidation outside.

reading (R) The datum is being read inside and is waiting for a response from outside.

exclusive responding (ER) The datum is exclusive, and responding to a read request from outside.

shared responding (SR) The datum is shared, and responding to a read request from outside.

Figure 2: Memory Controller Protocol

	read$_B$	write$_B$	read$_A$	erase$_A$	datum$_A$	erased$_A$
I	R:read$_A$	$*^3$	-	-	-	-
E	-	-	S:datum$_A$	*	*	*
S	-	W:erase$_A$	S:datum$_A{}^1$	I: -	-	*
W	*	*	-	I:write$_B{}^4$	-	E:write$_B{}^5$
R	*	*	-	R:read$_A{}^2$	S:datum$_B{}^6$	R:read$_A{}^2$

Notes 1: Only if selected by the bus arbiter.
2: Retransmit the 'read' request.
3: A 'write' miss is treated as a 'read' followed by a 'write'.
4: Reperform the 'write' as for a 'write' miss.
5: Proceed with the 'write'.
6: Proceed to store the 'datum' to complete the 'read'.

The following transactions may occur on buses:

read(X) A request to read the item X.

erase(X) A request to erased the item X.

datum(X,V) A response to a read request showing item X has value V.

erased(X) A response to an erase request showing item X is erased.

In addition, the following transactions may be observed between a processsor and its memory:

read(X) A request to read the item X.

write(X) A request to write the item X.

The above transactions, apart from a read request on a bus, have only a single phase. A read request on a bus, however, requires an additional arbitration phase in case many sibling controllers are willing to accept the request. We assume that each controller can observe transactions occurring on the bus above and respond if needed without input buffering. However, input buffering is needed for transactions on the bus below, and output buffering is required for transactions on either bus. The processing of transactions occurring on the higher bus may require examination of the output buffer above and resetting of some transactions to eliminate race conditions. One such situation occurs when erase(X) occurs on the bus above and an erase(X) request is stored in the output buffer above.

3.1 Memory Controller Protocol

The memory controller protocol is summarised in Figure 2. It shows, for each input transaction and state of the corresponding datum, the new state and output transaction. The symbol '-' indicates that no action is required and '*' means that the situation is impossible. The suffix A indicates a transaction on the bus above and B indicates a transaction on the bus below (which, for a memory controller, means simply the processor/memory interface). The memory controller performs the following functions.

If a read occurs below and the item exists in memory (a "read hit"), the read is simply allowed to proceed. Otherwise a "read miss" occurs, in which case the read request is transmitted to the bus above and the item is installed in the memory with status R. The replacement algorithm may need to be invoked.

If a write occurs below and the item is exclusive, the write is simply allowed to proceed. If the item is shared, an erase request is transmitted to the bus above and the item is marked W. If the item is invalid, this "write miss" is treated as a read miss followed by a write; of course this procedure can be optimised.

If a read request appears above and the item exists below with status exclusive or shared, the controller will respond and will transmit the item's value onto the bus above. If many controllers respond, the bus arbiter will select just one of them.

Figure 3: Directory Controller Protocol

	read_B	erase_B	datum_B	read_A	erase_A	datum_A	erased_A
I	R:read_A	-	-	-	-	-	-
E	-	E:erased_B	-	ER:read_B	*	*	*
S	-	W:erase_A	-	SR:$\text{read}_B{}^1$	I:erase_B	-	*
W	-	*	*	-	I:$\text{erase}_B{}^6$	*	E:erased_B
R	-	*	*	-	R:$\text{read}_A{}^2$	S:datum_B	R:$\text{read}_A{}^2$
ER	-	ER:$(\text{erased; read})_B{}^3$	S:datum_A	-^{4}	*	*	*
SR	-	W:erase_A	S:datum_A	-^{4}	I:erase_B	S: -^{5}	*

Notes 1: Only if selected by the bus arbiter.
2: Retransmit the 'read' request.
3: Transmit 'erased' followed by 'read' on bus below.
4: Should preferably be selected by the bus arbiter.
5: Cannot arise if bus arbiter selects ER or SR preferentially.
6: Output queue above is examined to remove any conflicting erase.

If an erase request appears above and the item exists below with status shared, the item is made invalid. If the item has status R, the read request is retransmitted above. If the item has status W, the write is reperformed as though a write miss had occurred.

If a read response appears above and the item is in state R, the value is stored in memory.

If an erased response appears above and the item is in state W, the write is allowed to proceed. If the item is in state R, the read request is retransmitted.

3.2 The Directory Controller Protocol

The directory controller protocol is summarised in Figure 3 with the same conventions as for the memory controller protocol. The directory controller performs the following functions.

If a read request appears below and the item does not exist within the subsystem (its status is invalid), the read request is sent up.

If an erase request appears below and the item is shared or in state SR, the erase request is sent up, and the item is marked W. If the item is exclusive, an erased response is sent down. If the item is in state ER, an erased response is sent down followed by a repeat of the read request.

If a read response appears below and the item is in state ER or SR, the response is sent up and the item is marked as shared.

If a read request appears above and the item exists below with status exclusive or shared, the controller will respond and, if selected by the bus arbiter, will send the request down. The item's status is changed to ER or SR according to its original status. If the original status was ER or SR, the controller will respond and will be preferentially selected by the bus arbiter; no further action is required.

If an erase request appears above and the item exists below with status shared, W or SR, the erase request is sent down and the item is marked as invalid. Any erase requests in the output queue above are removed to prevent race conditions. If the item is in state R, the read request is sent up again.

If a read response appears above and the item is in state R, the read response is sent down and the status is changed to shared. If the item is in state SR (which can't arise if the bus arbiter preferentially selects responders in state ER or SR), the state simply changes to S.

If an erased response appears above and the item is in state W, the erased response is sent down and the item is marked as exclusive. If the item is in state R, the read request is sent up again.

3.3 The Replacement Strategy

A basic philosophy of DDM is the absence of home address for data items. Instead of sending items to a home address that might be physically distant, a processor with a data overspill will tend to use the memories in his physical neighbour. This makes the machine scalable in a wider sense. In particular we achieve memory scalability, and locality zones are extended from being a single memory to a zone of adjacent memories. One can, if desired, trade processors for memory. This of course makes the replacement algorithms a bit more complicated.

The replacement algorithm consists of three operations: purge-up requests where replaced data items move up in the hierarchy, injection requests where purged up data items move down to eventually reside in some memory, and finally purge-down requests which are needed when a directory has no space for an item requested by some processors in its lower subsystem. The purge-down operation is needed because of the lack of perfect associativity of the directories. It forces some items to be removed from a subsystem to create space in the directory for items coming from outside. A full account of the replacement strategy can be found in a separate report [3]. Here we just outline the strategy, and, for the sake of simplicity, will ignore the purge-down operation.

Replacement of a data item occurs at the memory controller level, when a read miss occurs and the requested data item has no room. In this case, a purge-up request is made to free space followed by a read request. Purge-up requests at the memory level may eventually lead to purge-up requests at the directory levels.

Each directory controller has associated with it a small associative memory that is used to store a few data items that are either being purged up from the lower subsystem, or injected into it from the next higher level controller or from a sibling controller. We call such a memory the level buffer. A level buffer is used to approximate the degree of available space of the subsystem rooted at the current directory controller. This is indicated by a threshold Tlevel, which controlsthe number of items in the level buffer, denoted by Icount. Items in a level buffer may reside there, be sent up or sent down according to the following strategy.

Resident items: items purged up from the lower subsystem will reside in the level buffer as long as the number of items in the level buffer is less than Tlevel.

Items moving up: whenever Icount exceeds Tlevel, a number of items in the level buffer are purged up by requesting purge-up cycles on the higher bus; this will lead to a decrease of Icount.

Items moving down: if an item is accepted from the higher bus then the item will be temorarily stored in the level buffer, until it is injected into the lower subsystem; an injection cycle is requested on the lower bus.

4 MACHINE BEHAVIOUR

The data that a processor creates itself will automatically reside in its own memory. So long as no other processor request the data, the processor that created it can access it without causing any bus traffic. This is likely to cover the vast majority of data accesses. When a datum is created by one processor and subsequently accessed by another, the datum only needs to be copied over once. There is no need to repeatedly access a remote datum from its home location, as in most machines. In particular, once a processor has acquired some read-only data (e.g. program code), it can retain it in its local memory and need never again access the remote copy. If two nearby processors request the same remote datum, one of the processors can obtain it from its neighbour without the need for fetching it twice from the remote location.

A remote read takes at most 4N-2 bus transactions on an N-level machine (2N-1 read requests passing up to the topmost bus and down to the data, and the same number of read responses passing in the opposite direction). For example, there would be at most 10 transactions on a 3-level machine. To make a datum exclusive (in order to perform a write) an erase request goes up to the directory controller level where the item is exclusive; the directory controller acts as an arbiter of any competing erase requests, and sends an erased response downward. Thus (provided there is no competing erase request) the datum becomes exclusive after at most 2N transactions on an N-level machine (N erase requests passing up and N erased responses passing down). For example, there would be at most 6 transactions on a 3-level machine.

In general, the protocols have a combining effect for read requests going up, similar to that provided by the IBM RP3 multiprocessor [6], and a broadcast-data effect when read responses are going down, thus eliminating the "hot spot" phenomenon of the RP3. Thus in a 3-level machine, if one processor has a datum and the remaining processors request the same datum more or less simultaneously, all processors will get the datum in no more than 10 bus transactions.

In general remote data accesses only cause traffic within the subsystem concerned. For a read, only buses on the path between the source of the request and the source of the data are involved. For an erase, only buses on paths from the source of the write to copies of the data are involved.

The machine is scalable because there can in principle be any number of levels in the hierarchy. Note that the data access protocols are completely independent of the number of levels in the machine (and of the number of subsystems per bus). In practice there can be quite a few subsystems per bus (e.g. 16), so only a few levels are necessary to support a very large number of processors.

The hardware organisation of the DDM was partly influenced by a proposal of Hermenegildo [4] to provide an address escaping mechanism in clustered shared memory architecture (essentially a hybrid between a shared memory machine and a message passing machine). The DDM has many similarities to Wilson's proposal [9] for a hierarchical shared memory architecture, and certain similarities to the Wisconsin Multicube [1]. However all of these machines, unlike the DDM, depend on physically shared memory providing a "home" location for data. The Wisconsin Multicube can also be contrasted with the DDM in that certain requests need to be broadcast througout the entire machine.

5 OTHER ISSUES

Here we discuss some of the many issues that must be addressed in order to turn the basic idea of the DDM into a complete and fully functional machine.

5.1 Machine Configurations

To get a picture of possible configurations of the machine, we will make the following assumptions, which are intended as plausible approximations for the sake of discussion. We assume there may be up to 16 processors or subsystems on a bus; each processor has 1 M words of memory and delivers 250 K lips or 12 mips ; the cost of the machine per processor is $4,000. The maximum configurations of the machine for different levels corresponding to the numbers of levels in the bus hierarchy are then as follows:

Level	CPUs	Mwords	lips	mips	$
0	1	1	250K	12	4K
1	16	16	4M	200	64K
2	256	256	64M	3K	1M
3	4K	4K	1G	50K	16M

Thus, on these assumptions, a 2-level machine would be all that is needed for most purposes, while a 3-level machine might be considered to be the practical limit, supporting up to 4,000 processors and providing a total of 1 Glips. The physical memory on the latter machine somewhat exceeds the size of a 32-bit virtual address space. Any larger machine would almost certainly require a bigger address space.

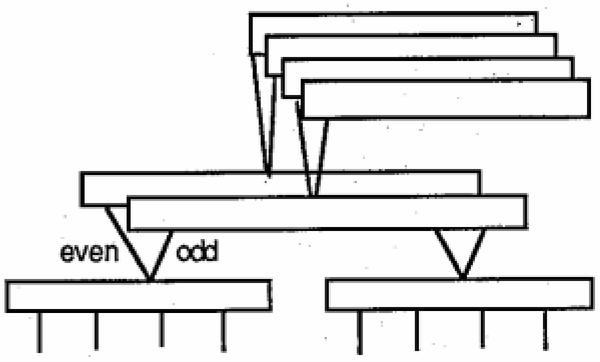
5.2 Size and Overheads of Memories and Directories

An important question is whether it is feasible to store in the higher directories the exact status of all the words below, or whether the higher directories should maintain only lower resolution information based on blocks of words. It appears to be feasible to store exact information. Assuming that memories and directories are 4-way set associative, the item size is one word, and that each level-1 memory contains 1 M words, each memory or directory at a given level contains the following:

Level 1: 1 M items with 3-bit status, 12-bit key, 32-bit value,
Level 2: 16 M items with 3-bit status, 8-bit key,
Level 3: 256 M items with 3-bit status, 4-bit key.

where the keys are the high-order address bits that must be stored to support set-associativity. Thus in a 3-level machine, the overhead per 32-bit word of storing the extra status information and address keys in both memory and the higher directories is 3+12+3+8+3+4 = 33 bits. Effectively we are doubling the memory requirement in order to provide the DDM's complete flexibility of virtual to physical address mapping. This seems a tolerable price to pay.

Figure 4: Splitting Higher Buses



5.3 Sparse Arrays and Null Values

Because all memory is associative, the programmer has complete freedom in how to use the virtual address space. In particular, there is no space penalty for sparse arrays. Only those items which are in actual use consume storage. To get maximum benefit from this effect, a certain value (e.g. zero) should be treated as 'null'. Writing null to an address will cause that item to disappear from memory. Reading a non-existent item will yield the null value.

Sparse arrays will be particularly useful for the SRI and Andorra models, since they are just what are required for binding arrays. The present versions of these models go to some lengths to avoid unused locations in the binding array. Implementation would be considerably simplified if the binding arrays were exact shadows of the corresponding shared areas, without regard for which locations were variables, or whether those variables were unconditionally bound.

5.4 Locking

Some operations, depending on the nature of the processor, need to be performed atomically. For example if the processor provides a test-and-set instruction this will lead to a read-modify transaction being performed by the memory controller. A read-modify behaves like a write except that before the data item is overwritten the original value of the item is fed back to the processor. With such an implementation of read-modify, together with the general behaviour of the machine, it is possible to perform spin-locking locally without generating any traffic, as shown below:

```
Lock(X):
    Start: Flag := Test&Set X;
           if Flag = 0 then Exit;
    Loop:  if X=1 then goto Loop
                  else goto Start;
    Exit:

Unlock(X):  X := 0;
```

where `Flag` is a machine register and `Loop` causes local spinning until `X` is modified.

5.5 Broadening Higher Buses

Although most memory accesses will tend to be localised within the machine, the higher level buses may nevertheless become a bottleneck. However it is possible to make the higher level buses effectively as wide as is needed by duplicating higher level buses and controllers to deal with different parts of the address space, splitting first on odd and even addresses, and then on successively higher order bits of the address, as illustrated in Figure 4.

5.6 Offsetting Latency

The delay for remote memory accesses may significantly degrade performance in larger machines. A possible means to offset such latency is to use a processor which can switch very rapidly between several "lightweight" processes. Although not taken into account in the data coherence protocols presented in this paper, it seems feasible to modify the protocols so that a processor will perform a process-switch when it issues a remote memory access, in order to be kept busy. For this approach to be effective, more parallelism is required in the application. For example, if the processors would otherwise spend half their time waiting for remote memory accesses, twice the normal parallelism is required to keep the machine busy. The machine behaves as though it has twice as many processors of only half the normal speed.

5.7 Secondary Memory

A data diffusion machine may have one or more disks attached to it. Disks behave as secondary memory subsystems which can hold overflow data. Each disk will be dedicated to hold some portion of the virtual address space. It will have associated with it a directory containing those addresses within its scope which are invalid, i.e. those which have been written to externally and where up-to-date copies have not yet percolated back to disk. The disk manager aggregates data into pages, and keeps recently accessed pages in a buffer. It responds to read request for data within its scope that are not invalid, but it is given lowest preference by the bus arbiter. If selected, it will fetch the relevant page from disk if necessary. In a similar way, it can also respond to purge requests, accepting data within its scope, fetching and eventually writing back the relevant page as necessary. It may use a packing algorithm to condense sparsely populated pages when they are stored on disk.

6 CONCLUSION

The Data Diffusion Machine (DDM) is a scalable shared virtual memory multiprocessor where the location of a datum in the machine is completely decoupled from its virtual address. In particular, there is no distinguished home location where a datum must normally reside. Instead data migrates automatically to where it is needed, reducing access times and traffic.

The machine is scalable in that there may be any number of levels in the hierarchy. Only a few levels are necessary in practice for a very large number of processors. Most memory requests are satisfied locally. Requests requiring remote access generally cause only a limited amount of traffic over a limited part of the machine, and are satisfied within a small time that is logarithmic in the number of processors. Although designed particularly to provide good support for the parallel execution of logic programs, the architecture is very general in that it does not assume any particular processor, language or class of application.

In future work, we plan to refine the design of the machine to a lower level and to carry out detailed simulations to verify its behaviour both in general and particularly on the SRI and Andorra execution models.

7 ACKNOWLEDGEMENTS

This work forms part of an informal collaboration known as the Gigalips project. We thank the many colleagues involved in or associated with the project, for helping to contribute to these ideas. The work was supported in part by the U.K. Science and Engineering Research Council, under grant GR/D97757.

References

[1] J. Goodman and P. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, pages 442–431, IEEE, 1988.

[2] S. Haridi and P. Brand. Andorra Prolog–an integration of Prolog and committed choice languages. In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[3] S. Haridi and D. H. D. Warren. The Data Diffusion Machine data access protocols and replacement strategy. 1988. Internal Report, Gigalips Project.

[4] M. Hermenegildo and P. McGehearty. *Address Escaping and Reference Classification in the Design of a Cached, Multiple Cluster, Shared-Memory Architecture.* PP-SRS-Technical Memo 12, MCC, 1987.

[5] E. Lusk, D. H. D. Warren, S. Haridi, et al. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[6] G. Pfister et al. The IBM Research Parallel Processor Prototype (RP3). In *Proceedings of the 1985 International Conference on Parallel Processing, Chigago*, IEEE, 1985.

[7] D. H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[8] H. Westphal, P. Robert, J. Chassin, and J. Syre. The PEPSys model: combining backtracking, and- and or-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*, IEEE, 1987.

[9] A. Wilson. *Hierarchical cache/bus architecture for shared memory multiprocessor.* Technical report ETR 86-006, Encore Computer Corporation, 1986.

[10] R. Yang. Programming in Andorra-I. August 1988. Internal Report, Gigalips Project.