Systems", Digital Technical Journal no. 7, August 1988.

[EGG88] Eggers S. and R. Katz, "Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation," Proc. of 15th Intl. Symp. on Computer Architecture, Hawaii, June 1988.

[FUR87] Furber S. B and A. R Wilson,"The Acorn RISC machine - an architectural view", Electronics and Power, vol 33 no 6, pp 402-405 June 1987

[GOO83] Goodman J. "Using Cache Memories to Reduce Processor-Memory Traffic," Proc. of the 10th Intl Symp. on Computer Architecture, Stockholm June 1983.

[GOO88] Goodman J. and P.J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," Proc. of 15th Intl. Symp. on Computer Architecture, Hawaii, June 1988.

[GOT83] Gottlieb A., et. al. "The NYU Ultracomputer--Designing an MIMD Shared Memory Parallel Computer", IEEE Trans. on Computers, VolC-32, Feb 1983.

[HIL86] Hill M.D. et. al. "SPUR: A VLSI Multiprocessor Workstation," IEEE Computer, 19, 11 November 1986.

[KAT85] Katz R.H. et. al., "Implementing a Cache Consistency Protocol," 12th international Symposium on Computer Architecture, IEEE, 1985, pp. 276-283.

[MCR84] McCreight E, "The Dragon Computer System: An Early Overview," Tech. Report, Xerox Corp., September 1984.

[NGU88] Nguyen T.M, Srini V.P, and A.M. Despain, "A Two-Tier Memory Architecture for High-Performance Multiprocessor Systems", Intl Conf. on Supercomputing, St. Malo, France, July 1988.

[PAT81] Patel J. H, "Performance of processors-memory interconnections for multiprocessors", IEEE Trans on Computers, Oct 1981, pp 771-780.

[PAT82] Patterson D.A., Garrison P., Hill M.D., Lioupis D., Nyberg C., Sippel T.N. & Van Dyke K.S., "Architecture of a VLSI cache for a RISC", 10th Intl. Symp. on Computer Architecture, 1982.

[ROS85] Rose C.D, "Encore Eyes Multiprocessor Market," Electronics July 8, 1985.

[SAT80] Satyanarayanan M. "Commercial Multiprocessing Systems ," IEEE Computer, 13, 5, May 1980.

[SEQ84] Sequent Computer Systems, Inc. "Balance 8000" Technical Summary, Nov 1984.

[THA87] Thacker C. and L. Stewart, "Firefly: A Multiprocessor Workstation", 2nd Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp 164-172, ACM, October 1987.

[WIL87] Wilson A. W. Jr, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," Proc of 14th Intl. Symp. on Computer Architecture, 1987.

[WIR82] Wirth N., "Programming in Modula-2," Springer Verlag, New York 1982.

better (bars 1, 4, and 7), but due to the sparse nature of some of the accesses (e.g. marking every 29th element), the greater prefetch is sometimes wasted, and the improvement is not as great as might be hoped.

5. Discussion

Looking at the four sets of results together, we can try to find a machine configuration that is amongst the best for all programs. Single bus machines perform badly with the SOR program (1 and 2 transfer cases are bad), and with matrix multiplication in the 4-transfer case. Multiple bus machines with only single transfers perform badly for the highly sequential quicksort program. Thus we are left with the best all-round performers being a 4-bus machine with 2 or 4 single-word transfers on a miss, or a 2 bus machine with 2 dual-word transfers.

To help us to understand the factors influencing the running times; the processors, caches and memories gather detailed statistics about the bus accesses during the run. These are recorded in log files, and can be post-processed to display load balance, queue lengths, utilisations, hit rates, read/write ratios etc.

These programs represent only a few examples of the sort of behavior expected from the inner loops of parallel applications, but already some trends can be seen:

> A large transfer unit is detrimental to programs that write shared data interleaved at a fine grain, as it can lead to unnecessary invalidations. An example would be updating items in a shared job queue or other list structure, where a change to one element could invalidate adjacent entries that were being worked on by other processors. This will lead to bad performance on wide bus machines with multiple transfers.

> Buses with single transfers are not using the memory to best advantage (page modes, suppressed addressing of sequential blocks). Wide bus machines with single transfers are consistently slower than multiple bus machines with the same transfer unit. Compare the 1-bus 1-transfer, 2-bus 2-transfer, and 4-bus 4-transfer running times; all of these systems move data in four-word blocks, yet the 4-bus configuration is consistently faster.

From these observations, we expect that multiple bus machines will be worthwhile in environments where the very best performance of parallel algorithms is sought. They can provide high bandwidth to memory without the penalties of large transfer units, and with the added flexibility of concurrent transfers. When using wide bus machines for parallel applications, more care is required to make the best use of their potentially high bandwidth.

6. Conclusion

We have run a selection of parallel programs on nine variations of shared memory multiprocessor architectures. The results have been encouraging, as the special characteristics of each program have led to predictable differences in performance on the nine machines. Our original thesis, that multiple narrow buses offer a high bandwidth with more flexibility than single wide buses, has been borne out by the poor performances observed in some wide-bus runs.

Our simulations have all used small programs operating over a relatively small range of addresses. Whilst they cannot be taken as typical examples of complete multiprocessor applications, they are representative of the inner loops of compute-intensive programs. The instructions of such pieces of code are always cached, and often constitute the major part of the running time, but the data they access might not be cached. If they are writing to shared data, or are reading large amounts of data, then they will generate bus traffic similar to that seen on our simulated systems.

With current technologies, multiple bus machines of a significant size are not cost-effective. The crossbar interconnect of 32-bit buses, with high-speed cache elements at each intersection, proves very difficult to implement. As high-density interconnect systems become more widely used, the implementation of multiple buses will become easier, making such systems more attractive.

Wide-bus architectures, being more suitable for implementation on a conventional backplane, are now emerging in high-performance machines. It remains to be seen whether the unfavourable characteristics displayed by the SOR and matrix product programs can be avoided in practice.

Acknowledgements

The authors would like to express their thanks to James Kenney and Kami Sehat of Cambridge University Computer Laboratory for writing the processor and cache models; Stuart Wray and Mark Chopping of Olivetti Research Laboratory, Cambridge for their work during the development of the multiprocessor simulations and programs described in this paper.

References

[AGA88] Agarwal A. and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under MACH," Proceedings of ACM Sigmetrics 1988.

[ARC86] Archibald J. and J. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors," ACM Trans. on Computer Systems, 4,4, November 1986.

[BEL85] Bell C.G, "Multis: A New Class of Multiprocessor Computers," Science, 228, April 1985.

[CHE88] Cheriton D.R., A. Gupta, P.D Boyle and H.A Goosen, "The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation," Proc. of 15th Intl. Symp. on Computer Architecture, Hawaii, June 1988.

[DAS85] Das C.R, and L.N. Bhuyan, "Computation Availability of Multiple-Bus Multiprocessors", U of Southwestern Louisiana, 1985.

[DEC88] Digital Equipment Corporation, "CVAX-based

obtained by the simulator are shown in figure 3.

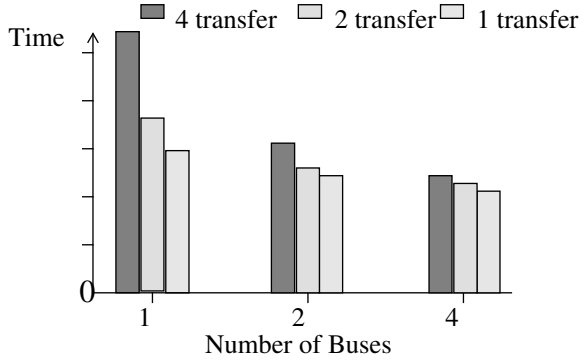

Figure 3 : Performance of Prod Program

There is a lot of read sharing in this algorithm, and fewer writes per instruction than the others, yet the average queue length in the wide bus case (single transfer, bar 3) was 6.45. This dropped to 1.93 and 0.92 in the 2-bus and 4-bus cases (bars 6 and 9). The main reason for the high contention on the wide bus is that each processor is responsible for computing every 16th element of the result. When one word is written, it invalidates the entire block, which contains the words being computed by three of the other processors, requiring extra bus transfers when they come to write their results. We therefore see this program being dominated by the effects of the wasted prefetch, with the smaller effect of the increased memory efficiency of multiple transfers playing a secondary role.

4.3. Quicksort

The quicksort algorithm has been rewritten for concurrent execution on the multiprocessor. It is still based on recursively dividing the input list into two lists with elements smaller than and greater than a pivot. The first element of the list is used as pivot. In the beginning only one processor starts executing and the rest spin on a lock. When the list is divided the processor keeps one half of the list and gives the other to the next processor by clearing its lock. This is repeated until there are no more processors, when the execution is reduced into normal recursive quicksort. The list is stored as a global array of random numbers in memory. The performance of this program depends heavily on the input list which determines the load balancing of processors. The program was run sorting 1000 random numbers and the results are shown in figure 4.

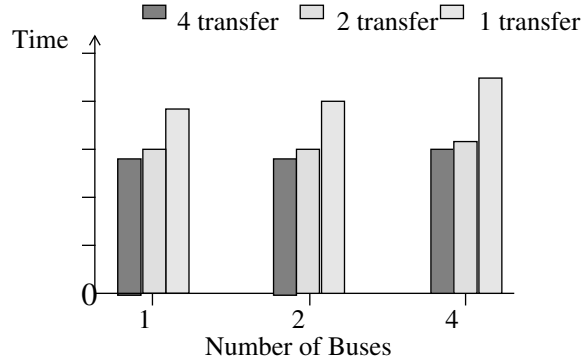

Figure 4 : Performance of Quicksort Program

Quicksort is the only program to take full advantage of the prefetch provided by the wide bus (bar 3), and runs slightly faster than with narrow buses in the single transfer mode (bar 9). When prefetch was added to the narrow bus case (bar 7), the program ran in only 70% of the time, outperforming the wide bus in all but the 4-transfer case (bar 1, 16 word blocks).

4.4. Sieve

This is a parallel implementation of the sieve of Eratosthenes. The main data structure is an array of 1024 integers, where the contents of array[i] indicate whether i is a prime or not. All processors start at the second element and move up the array looking for a zero. If they find one, they use a test-and-set instruction to mark the number as prime (a 1), and then proceed to mark all multiples as non-prime (a 2). When all processors have searched as far as the square root of the array size, the final processor scans the array, counting the number of primes (zeroes or ones).

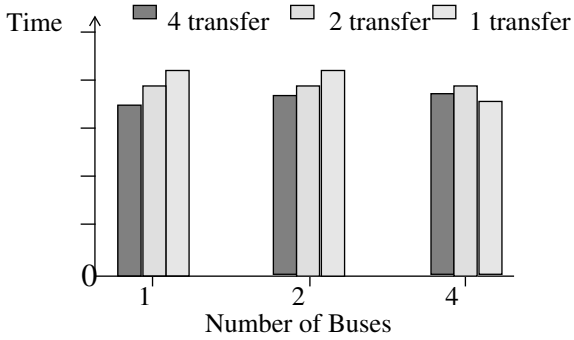

Figure 5 : Performance of the Sieve Program

Due to the restricted length of the sieve array, only eleven of the sixteen processors have any work to do, and after about 25% of the running time, the other five have found this and are idling. During the rest of the run, there can be considerable interference as processors rapidly mark off multiples of the remaining primes, so although the hit rate improves with prefetch, the invalidations also increase, leaving the running time largely unaltered. To show that bus bandwidth is indeed a bottleneck in this instance, we also show the mean queue lengths on the buses in figure 6.

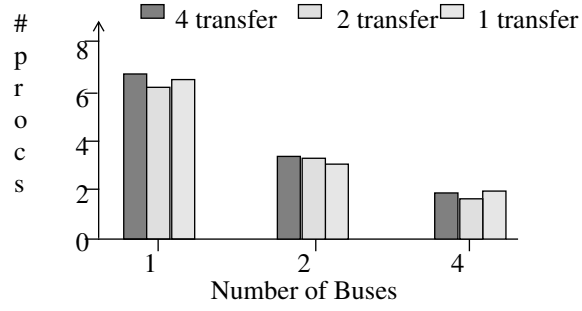

Figure 6 : Average Queue Lengths for the Sieve Program

Note that the queue length divided by the bus width remains roughly constant, so although the wide bus machines have greater latencies on a miss, their buses are transferring more data each time, and the overall performance becomes more a function of the efficiency of memory usage. We should therefore expect the multiple transfer machines to do much

table 1.

	Bus Width		
Transfers	1	2	4
1	1	2	4
2	2	4	8
4	4	8	16

Table 1: Cache block sizes (words)

4. Results

The results obtained are shown in the next four sections in bar chart form, one section for each program. Each bar represents the execution time of the program in microseconds (see figure 2). Thus a smaller bar represents a better processor performance. The text will refer to the bars by numbering them from the left, thus the wide-bus, 4-transfer case is bar 1, and the 4-bus, 1-transfer case is bar 9.

We also quote other statistics gathered from the simulations. Utilisation is given as the mean of the percentages of time that each individual bus is occupied by *any* processor. Average queue length is the mean number of processors that are requesting or have been granted the bus. We collect these statistics by observing the buses every 20ns., the period of the master clock from which all other clocks in the system are derived.

4.1. SOR

This program repeatedly computes the value of grid locations by taking the average of the four surrounding points. Eventually, this method converges to a solution of Laplace's equation for the given boundary values. Our grid consists of 32 rows of 9 elements each, and we perform 25 iterations. For multiprocessor execution, we divide the grid equally into as many horizontal strips as there are processors. Each processor then updates every other element in its strip in turn, and increments its own iteration count. It then spins, waiting for its two neighbours to come to the same iteration number, then proceeds with the next wave of updates. At the edges of the strips, neighbouring processors are always trying to read the same locations, so the sharing overhead is high.

The running times for this program are shown in Figure 2..

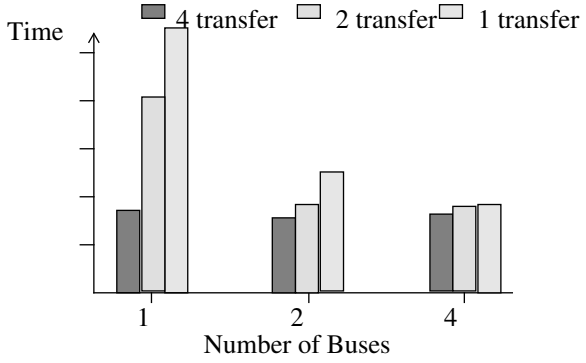

Figure 2 : Performance of SOR Program

Looking at bars 3, 6 and 9, the most striking feature of figure 2 is the very bad performance of the wide-bus machine with only one transfer per miss (bar 3). Normally, we would expect the prefetch given by a four-word block to enhance performance, but in this case it has dramatically reduced it. This is mainly due to the data for different processors being interleaved at one word grain in the regions where it is shared (a property of the software implementation). This leads to higher utilisation in the wide bus cases (greater block sizes): 99.8%, 89.7% and 41.6% for the 1, 2, and 4-bus cases respectively (bars 3, 6, and 9), leading to average queue lengths of 9.17, 3.49 and 0.88. The hit rates improve from around 90% on the narrow buses to 94% on the wide, not enough to overcome the effects of the much greater queue lengths.

The narrow bus machine with four transfers per miss (bar 7) does not suffer from this problem as it makes much better use of the memory (4 buses by 4 words = 16 words, transferred in 400 + 200 + 200 + 200 = 1000 ns.) than the wide bus with one transfer (bar 3, 4 words transferred in 400ns.). It also allows one processor to block its interfering neighbours for a longer period, but only on one bus, so the other buses become less congested, and other processors can proceed efficiently. Because the program causes a considerable amount of bus traffic, the buses become saturated if the memory is not used to best advantage, and we see 99.8% utilisation, 9.2 queue lengths for both the two- and single-transfer wide bus cases (bars 2 and 3). If we consider just the one-transfer cases, then we should see the effect of the greater prefetch as the buses become wider, as the time to satisfy a miss remains constant at 400ns.

4.2. Matrix Multiplication

The matrix multiplication program multiplies two matrices (16 x 16 elements) stored as global arrays to produce the result matrix. Each processor calculates a part of the resultant matrix determined by its number, which is used to index the resultant array. This means that eventually each processor will fetch all elements of the first matrix and a column of the second matrix. Write invalidations in the blocks that hold the result should influence the performance of this program. The results

3. Simulation

3.1. Simulator Details

We have developed a high-level event driven simulator to model various multiprocessor systems. Behavioural models of the individual chips and bus wires are written in Modula-2 [WIR82], and they accurately reflect the detailed timing of the external logic signals. For instance, the processor model fetches instructions from the memory model by driving signals in the same way as the real chip. The whole system is generated from a set of high-level parameters such as the number of processors, cache associativity and wire delays.

The underlying model is one of nodes (circuit elements) communicating by sending values (64 bits) across contacts. This allows us to pass 32-bit bus values in one event, and to display them in a meaningful way to the user (e.g. hexadecimal values). As the interpretation of these values is defined from outside the simulator, it can be tailored to the application, for instance, by disassembling instructions when the values on the data bus are displayed.

All the models register extensive debugging commands with the user interface. For example, breakpoints can be set on processor addresses to stop the simulator and allow register dumps or single stepping, the values stored in caches and memories can be read or written, and individual addresses can be monitored to trace all changes. When investigating the performance of some algorithms, it has proved particularly instructive to watch the accesses to a lock and the data items it protects.

To save time, the memory model can interpret loadable images, and initialise itself directly, removing the need for a loader and simulated input device. The processor modeled is the Acorn RISC Machine (ARM) [FUR87] with a cycle time of 200 ns and is connected by a bus translator chip to the caches. The caches are our own design, they are write back and communicate with the global shared memory over buses that implement the Berkeley consistency protocol.

Running on a Sun260 workstation, a simulated single processor machine runs at around 35 ips (instructions per second). Machines with larger numbers of processors impose a greater load on the simulator, keeping the aggregate instruction throughput at 15 to 30 ips. Our benchmark programs require around 1 million cycles to complete, and take between 8 and 24 hours for each run.

3.2. Programming Environment

The programs are written in C, with assembler libraries for booting, synchronisation and output. The main procedure takes two arguments representing the number of processors in the system, and the number of the processor that it is executing on. These numbers are computed by routines contained in the boot code. Each program can be linked for execution directly on the Archimedes workstation which compiles it, and debugged in single processor mode before being used in the simulation. After they have solved their particular problems, they use unimplemented instructions to signal the models to report statistics for that run, and they then proceed to verify that the results were correct. If the answers are correct, then a one line summary is entered in a collation file from which various performance graphs can be drawn. If the answers are incorrect, then the time is entered as zero to draw our attention to it, and the entry is ignored by subsequent tools. These checks have detected subtle errors in our C test programs, and faults in our simulated hardware that only rarely occur in particular configurations. The correct answers are obtained by running the programs on the real ARM in the Archimedes workstation.

3.3. Simulated Machines

We simulated four simple programs runs on three bus layouts to compare their performance:

1) Wide-bus: A 128 bit wide bus with 16 processors.
2) 2-bus: Two 64 bit wide buses with 16 processors.
3) 4-bus: Four 32 bit wide buses with 16 processors.

All systems use caches as shown in figure 1. The cache size is 1 kbyte per processor, which means that in the 4-bus case each cache is 256 bytes. It is small to correspond with the small size of our programs. The total system cache is also constant because we are using 16 processors. The buses use separate paths for addresses, data, and control, and the master keeps control of the bus until its request is satisfied. Arbitration for the bus takes 50 ns. and a memory fetch 400 ns. (plus 200ns. for each additional word).

The cache block size is important in a shared memory architecture because it influences the amount of traffic on the bus and thus contention. For each of the above bus configurations we performed three runs of each program to determine the effect of block size on our measurements.

Single Transfer

The cache block size is equal to the bus width. In this case, on a cache miss, a block will be transferred in one bus cycle.

Two-transfer

The block size is now twice the bus width. Whenever data has to be moved between cache and memory, one address is sent, followed by two cycles of data transfer.

Four-transfer

The block size is four times the bus width. Each block will be fetched in four sequential memory transfers. We gain some advantage by only presenting the address information once, and by using memories with fast page modes. In the *Wide-bus* case we are fetching 16 words, which may be advantageous for some programs, and not for others. The cache block sizes (also the data transfer sizes) for the nine runs are shown in

Each memory module contains a portion of the memory space and memory references are interleaved in cache block size intervals. If the cache block size is four words, then addresses 0 to 3 will reside in the first memory module, 4 to 7 in the second and so on. The caches implement the Berkeley protocol [KAT85] to maintain consistency on each memory bus by ownership and snooping. Because each portion of the address space (e.g. 0 to 3) is always mapped onto the same memory module, consistency can be maintained independently on each memory bus by the snooping mechanism in the corresponding caches.

The resulting architecture maintains the same programming model as conventional shared bus multiprocessors, whilst providing a higher bandwidth to memory. The number of memory buses is limited by the electrical load on the processor bus to about 4-8. We simulated up to 4 memory buses.

As we increase the number of memory buses, there is a linear increase in the number of cache chips, and in the number of buses. Many organisations, including Olivetti, are working on high density silicon-based interconnection technologies which could be used to implement such crossbar systems.

Recently many architectures have been proposed utilising both wide bus and multiple bus approaches. They all address the shared memory bottleneck problem by increasing the width of the data path to memory.

A multiprocessor with a 128 bit wide bus is under investigation by Olivetti. It is designed to accommodate up to 8 processors connected through a write back cache onto a pended bus. A processor issues a request, and releases the bus, then the corresponding memory module requests the bus when it has the reply ready. The bus operates at 20 MHz and can be extended by connecting two similar buses with a special interface. In our simulations we study a similar architecture which employs a write back cache, but with a different policy (Berkeley ownership). We use a master slave bus model which will behave differently to a pended model. Our multiple bus case however, shares the ability to have several transactions in progress at the same time, and these results may relate to the pended model.

The Wisconsin Multicube[GOO88], is a shared memory multiprocessor which uses a grid of buses to connect to memory. In this design there is a processor at each cross point resulting in a large number (up to 1024) of processors. We envisage a smaller number (up to 100) of processors connected to memory through caches at each cross point as shown in figure 1b. This simplifies the cache consistency mechanism which is a major problem in the Multicube. Consistency checks occur independently on the vertical memory buses, as in a conventional single bus multiprocessor.

Multiple buses are used on Aquarius as reported in [NGU88]. Processors are connected to memory buses through caches at each cross point. Aquarius is a multiprocessor designed for Prolog, which has a different model of execution to conventional languages, and imposes higher demands on the memory system. Due to the increased number of memory writes, separate buses are used to carry the invalidations to other caches. Its designers believe that multiple buses can provide enough bandwidth to meet these demands. We concentrate on the parallel execution of conventional languages and in particular programs written in C.

Factors Affecting the Performance of a Program.
This section discusses some of the factors that can affect the performance of multiprocessor systems. The different running times of a program on various configurations may not be directly related to the cache/memory bandwidth. The cache/ memory parameters can subtly alter important factors such as load balance, synchronisation, or the detailed access pattern of the program. The programs that we use were chosen to depend differently on such factors.

As a concrete example for this section, we shall assume that 128 wires are used to carry the data on the memory bus, organised as a single transfer unit in the wide bus case, and as four independent 32-bit word units in the multiple bus case. The processor to cache bus is assumed to operate with 32-bit quantities.

On a cache miss, the wide bus case is assumed to transfer four sequential words, aligned on a four-word boundary. If these four words are not all useful to the processor during their time in the cache, then some of the extra bandwidth provided by the bus width is being wasted. However, other studies show that a few words of prefetch generally improve the cache hit rate, and that this outweighs any wasted data fetches [DEC88].

When an algorithm is designed to operate on a number of processors, it is often difficult to avoid one or two of the processors having to do more work than the others. In some cases this imbalance of workloads may be dependent on the data supplied for the run(e.g. quicksort). In others the imbalance may be inherent in the chosen implementation (e.g. a prime number sieve). In almost any parallel algorithm, there will be some computation that cannot be performed concurrently, and this will further upset the work distribution. Under these circumstances, some processors will become idle, leading to longer running times, but less bus traffic. Our program suite includes two such programs, but the amount of idle time is such that a 16 processor system still completes them faster than any less powerful machine.

In most programs, explicit synchronisation between processes occurs infrequently compared to the time spent in other computations [EGG88]. However, after processes have synchronised, they may for a time run together through the same data structures (e.g. a work queue) and this can lead to beneficial or detrimental interference until the processes move apart. Changing the memory parameters can subtly alter the amount of time that the processes are interfering, and this can have noticeable effects on performance.

Changes in the layout of data, or in the times that it is accessed during a computation, can drastically change the running time. The classic example of this occurs when scanning through matrices. If a cache block contains consecutive elements from the same row, then running through a row will take maximum advantage of the prefetch mechanism, whereas a column will only use one word per block.

Multiple vs Wide Shared Bus Multiprocessors

Andy Hopper, Alan Jones, Dimitris Lioupis

Olivetti Research Ltd.
Keynes House
24A Trumpington Street
Cambridge CB2 1QA
England

ABSTRACT

In this paper we compare the simulated performance of a family of multiprocessor architectures based on a global shared memory. The processors are connected to the memory through caches that snoop one or more shared buses in a crossbar arrangement.

We have simulated a number of configurations in order to assess the relative performance of multiple versus wide bus machines, with varying amounts of prefetch. Four programs, with widely differing characteristics, were run on each configuration. The configurations that gave the best all-round results were multiple narrow buses with 4 words of prefetch.

1. Introduction

Multiprocessors are used today to provide better performance at lower cost. Many commercially available systems are based on a shared memory, shared bus architecture. These machines have a relatively straightforward implementation since they are an extension of the uniprocessor bus system. Their globally shared memory and consistency mechanisms give a programming model that is very similar to systems of cooperating processes on uniprocessors. Commercial systems such as the Encore [ROS85] and Sequent [SEQ84] claim significant speed-ups at very low cost.

A major limitation of shared bus multiprocessors is the bandwidth of the bus, which limits the number of processors that can be connected to the same memory, and thus the performance of the system. To solve this problem we can increase the speed of the bus [DEC88], which is not always easy because of technology limitations, or we can use more wires to connect to the memory. For a given technology, more wires provide more bandwidth, but it is not obvious which is the best way to connect the wires because of complications such as caches, code sharing and system complexity.

Wide buses are simpler to build but they provide only one path to memory. Multiple buses are more complex to implement but they reduce contention because of multiple paths to memory and more wires for control and addresses.

In this paper we study the effect of bus architecture on performance. Keeping the number of data wires constant, we found that multiple buses can provide better effective bandwidth to memory, and thus better performance. Multiple bus machines ran our sample programs from 0.9 to 3.5 times faster than wide bus machines.

The rest of the paper is organised as follows. In the next section we give a brief description of the proposed architecture and we compare it with existing designs. In section 3 we review the factors that may affect program performance to aid in understanding the results. Section 4 describes the simulator environment used to obtain the results included in section 5. We conclude with an overall analysis in section 6.

2. Architectural Description

Conventional multiprocessors (such as SPUR [HIL86], Firefly [THA87], and others) are connected to the shared memory by a single bus as shown in figure 1a.

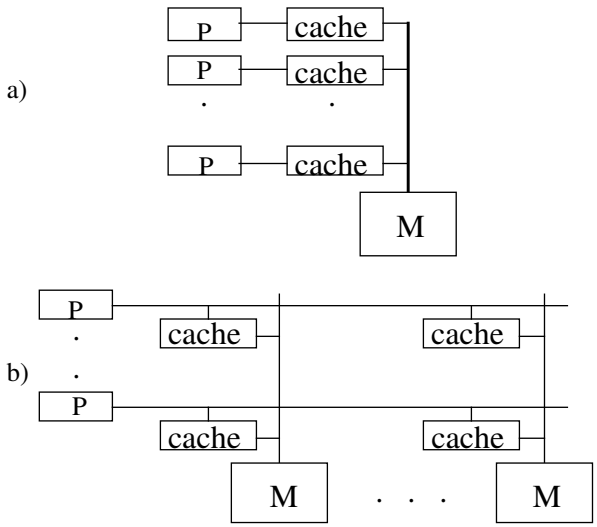

Figure 1 : Shared bus multiprocessors

To increase the bandwidth to memory, we use a multiple bus architecture. The resulting architecture shown in figure 1b, uses a grid of buses with a cache at each cross-point to connect to memory.