DESIGN AND IMPLEMENTATION OF AN ORDERED MEMORY ACCESS ARCHITECTURE[1]

S. Sriram, and E. A. Lee

EECS Department, UC Berkeley, Berkeley CA94720.
(sriram@ohm.berkeley.edu, eal@ohm.berkeley.edu)

December 15, 1992

— *Proc. of ICASSP '93*, Minneapolis, April, 1993 —

ABSTRACT

This paper describes a multiprocessor machine for real-time Digital Signal Processing that uses commercial programmable DSP chips. The architecture is a shared memory, single shared bus parallel processor designed to run signal processing tasks that can be statically scheduled. The design is based on the architecture proposed in [1]. A prototype has since been built. The implementation details and performance results are discussed here.

1 REAL-TIME SIGNAL PROCESSING USING MULTIPLE DSP CHIPS

DSPs are ideal for medium throughput applications, such as speech and digital audio. However, when attempting to meet real time constraints, one is often faced with the task of squeezing the application to fit onto relatively few instructions available per sample on a processor.

For example, consider processing audio at a 44 KHz sampling rate on a processor with a 60 ns cycle time. Audio samples arrive once every 227 ms. Hence there are only about 380 instructions available to process each sample while maintaining real time constraints.

Use of multiple DSPs is thus an attractive option for increasing the amount of processing that can be done per sample. The utility of a multi-DSP machine is determined largely by two issues. First is the overhead associated with communication and synchronization between processors and second is the available software environment for programming the multi-processor machine. Several multi-DSP architectures have been demonstrated by the research community. Special hardware is usually employed for reducing communication overhead. The SMART array [2] for example is a distributed shared memory machine with custom VLSI parts for maintaining coherence. The iWARP architecture [3] employs separate communication and computation agent, implemented using custom VLSI. In [4], the authors describe a DSP96002 based multi-DSP system with an "intelligent communication controller" implemented on a gate array for communicating through a high speed bus.

Our approach is to reduce overhead by restricting ourselves to a certain subclass of DSP applications. By sacrificing generality, we acheive efficient communication between processors with relatively simple hardware. Such an approach is ideal for low cost implementation of embedded applications that fall into this subclass, on multiple DSPs. Also, software environment used for programming the multi-processor machine is closely tied to the hardware methodology employed.

2 SYNCHRONOUS DATA FLOW

We address DSP applications that can be specified as a data-flow graph, with nodes (actors) being individual tasks and directed arcs between them representing flow of data (tokens). Synchronous Data Flow (SDF) refers to a subclass of data flow graphs where the actors lack data dependency in their firing patterns [5]. Thus in SDF graphs the number of tokens produced and consumed in each of the output and input arcs of each actor is constant and fixed at compile time. In exchange for this restriction several nice properties are obtained for these graphs. In particular, self-timed scheduling, i.e. assigning actors to processors and specifying their firing order, can be done efficiently at compile time. Nearly optimal static periodic multi-processor schedules can be obtained for these graphs [6]. Also, given approximate execution times for the actors, one can constrain the order in which processors would need to access shared resources at run time without unduly sacrificing performance.

It is seen that a large set of DSP algorithms fall into the SDF paradigm. In our group at Berkeley, we have implemented Gabriel, a block diagrammatic software environment for DSP based on the SDF model [7]. Ptolemy, the next generation system developed by our group, handles several different models, SDF being one of them [8]. Both systems can generate executable code for processors from block diagram specifications. Parallel scheduling algorithms have been implemented to schedule a flow graph onto multiple processors. Each processor gets a subset of all the nodes present in the flow graph, and it executes them in the prescribed order. When data tokens need to flow between processors, interprocessor communication

1. This work was supported by grants from SRC (grant number 92-DC-008), Motorola Inc., and ArielCorp.

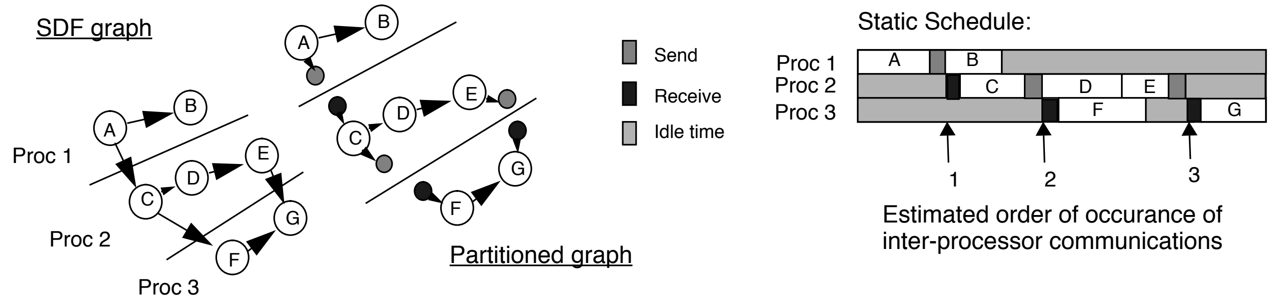


FIGURE 1. Illustration of the scheduling mechanism

primitives are inserted in the code of the sender and the receiver. The dataflow graphs we deal with are typically fine to medium grained, with each node representing no more than few tens of instruction cycles. Data going from one processor to another is usually one word at a time, in other words the data is not ‘packetized’ into blocks. Figure 1 illustrates the multiprocessor scheduling mechanism.

Under the software environment above we describe an architecture for synchronous data flow.

3 ARCHITECTURE

3.1 Motivation

Our goal was to design parallel hardware that would mesh well with the software methodology to be used on top of it. The result, hopefully, would be an architecture optimized, both in terms of performance as well as cost, for the restricted set of applications (in this case SDF) that we were concentrating on. We look at a single bus shared memory architecture, mainly because it results in simpler hardware and an easier partitioning problem. For such an architecture, communication of data through shared memory involves bus and memory contention. Usually there is a bus arbiter to resolve simultaneous bus access requests. Memory accesses, then, must be synchronized using semaphore mechanisms. All this adds to the inter-processor communication (IPC) overhead, not to mention the bus bandwidth wasted on unsuccessful semaphore checks.

In our lab we had a four processor Motorola DSP56000 architecture based on the “bus arbiter and semaphore” mechanism. Because of the associated overhead, 30 instruction cycles were required to transfer a word between processors. For digital audio, 30 cycles represents about 8% of the processing time available on a processor, in the scenario of Section 1. This implies that the scheduler must use as few IPCs as possible, thus severely restricting the parallelism that could potentially be exploited.

In principle we could reduce the IPC overhead by managing semaphores in hardware, using multi-ported memory and so on, under the penalty of complex and expensive hardware. Instead, noting that the above approach is too general for our purposes, we look at an alternative solution.

3.2 Ordered Transaction Principle

Suppose we knew the exact run time behavior of each of the task nodes, down to the clock cycle. Also suppose that all processors run synchronously. We could then determine at compile time the instruction each processor executes on each clock tick (the *fully-static* approach in [1]). This approach would obviate bus-memory contention and result in near zero overhead IPC. However it would be impractical, since we usually do not know execution times of nodes to the degree required here.

Now, suppose we retain only the *order* of execution of nodes and the *order* in which shared memory is accessed, based on the static schedule. Since the application fits the SDF model, imposing this constraint does not sacrifice efficiency, provided that approximate execution times of the nodes are known. The approach is robust even for inaccurate execution time information. Performance efficiency however, may suffer in this case. Run time imposition of an order pre-determined at compile time obviates bus and memory contention, as in the fully-static case. No bus arbitration is required, and semaphores are not needed. A central controller simply maintains the pre-determined order by granting the bus to each processor on its turn. In the best case, if the scheduler has done a good job, the processor already has the bus when it needs it and can go ahead with its shared memory transaction. On the flip side, if the scheduler does not do a good job, a processor may have to wait until the bus is granted to it, or even worse, a processor may not yet be ready when it is granted the bus, thus blocking other processors that need the bus.

This approach is hardware lean: bus arbitration logic already present on the processors can be used. The access controller is simply a counter addressing memory that is downloaded with the schedule at compile time. No other hardware is required to implement the scheme. Moreover, in the best case it takes only 3 instruction cycles per transaction, which is about as well as we can hope to achieve.

3.3 Hardware Implementation

As a proof-of-concept prototype, we have built a 4 processor single bus shared memory architecture, called the ordered memory access architecture (OMA) board, with hardware support for ordering shared memory transactions (Figures 2 & 3). In our implementation, we use 4 Motorola DSP96002 processors on a single printed circuit board, operating with a 33MHz common clock. Even though our techniques can be applied to most off-the-shelf DSP chips, we chose the DSP96002, because its dual bus 32-bit architecture lends itself naturally to our specific application. A processing element consists of a DSP96002 with up to 256Kbytes of onboard zero wait state static RAM on one port (local memory); the other port of all processors are connected together to form the shared bus. There is provision for up to 512Kbytes of onboard static shared RAM.

The transaction controller is implemented as a presettable counter on a Xilinx FPGA. This was done with a view towards future experimentation with the transaction ordering mechanism. The access order list is extracted from the schedule and downloaded into the transaction controller memory. The schedule counter steps through this list at runtime. Stored entries in the schedule RAM correspond to processor IDs. This output is latched and decoded to obtain bus grant signals for the processors. The schedule counter is incremented each time the bus is released by a processor. Host interface and a simple I/O mechanism is also implemented on the Xilinx array. Re-use of the gate array has saved us a fair amount of glue logic, thus making the board less complex. Buffers for connecting multiple boards are included. We can configure multiple boards such that they form a single bus, or we could have cleaved busses with communication between busses implemented on the Xilinx chip.

The host to this board is a DSP56000 Sbus based card from Ariel Corp. This card is used to control the OMA board as well as download code from a SPARCstation running UNIX. The 56000 also acts as a interrupt processor, providing the OMA board with real time I/O. Currently, we feed the OMA board with data from a compact disc (CD) player and obtain data from it at CD rates.

The OMA board is a 10 layer through-hole PCB, with dimensions of 11” by 7”. It was designed and laid out using the tools developed here at UC Berkeley [9] and was fabricated by MOSIS.

4 RESULTS

The 4 processor prototype has been tested and functional correctness has been verified. We have been able to achieve processor to processor to communication over the shared bus with a cost of 3 instruction cycles. The multiprocessor system has also been integrated into Ptolemy and Gabriel, for automatic scheduling, code generation and downloading from a Sun Sparc workstation. Some of the applications that have been tested on the OMA board are as follows.

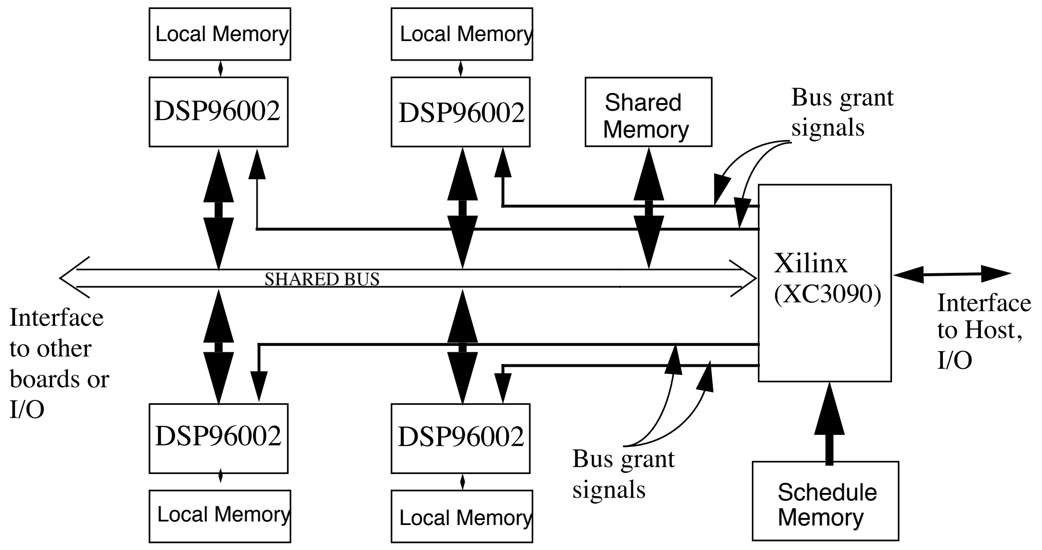
4.1 1024 point complex FFT

Data is assumed to be present in shared memory. The transform coefficients are written back to shared memory. A single 96002 processor on the board performs the transform in 3.0 milliseconds (ms). With all four processors, it takes 1.0 ms. Each processor performs a 256 point FFT from the first stage of a radix 4 computation. This example is communication intensive: the throughput is limited by the available bus bandwidth.

4.2 Music synthesis

A synthesis algorithm for plucked strings was implemented using the Karplus-Strong algorithm. Synthesis of each voice involves a noise source, a pulse generator, a delay, a filter operation and scaling operations. A single processor could fit 7 voices in real time (44 KHz sampling

FIGURE 2. Block diagram of four processor OMA board



rate). Partitioned accross 4 processors, with 15 IPCs, we could fit in 28 voices. This example is not communication intensive; the low overhead IPC mechanism easily absorbs the extra cycles associated with IPC.

4.3 QMF filter bank

A filter bank was implemented to decompose audio from a CD player into 5 bands. The resynthesis bank was also implemented together with the decompostion part. This involved 16 multirate filters (18 taps each). There were 85 IPCs in the final schedule .

5 CONCLUSION

We have presented the design and implementation of a multi-DSP architecture that acheives low overhead interprocessor communication with low hardware complexity. Program entry, scheduling and code generation for this can be done under the Ptolemy environment. Some applications have been run on the board. We plan to expand the existing I/O capability of the board by adding peripheral modules. This will enable us to evaluate the architecture under several other applications.

Imposition of a single fixed order at run time means that no data dependency can be tolerated. To run non-SDF graphs on the OMA architecture, we use a presettable counter as the transaction controller. Any processor that has posession of the shared bus can make the transaction controller jump to another bus access schedule by presetting the schedule counter. Conditional branches can thus be handled by computing access schedules for each branch outcome and switching between them at run time. Compilation process for this scheme under the Ptolemy environment will be the subject of future research. We can then evaluate how well this approach to handling limited data dependency works in practice.

6 REFERENCES

[1] J. C. Bier, S. Sriram, and E. A. Lee, “A Class of Multiprocessor Architectures for Real-Time DSP”, *VLSI Signal Processing IV*, 1990.

[2] W. Koh, “A Reconfigurable Multiprocessor System for DSP B ehavioural Simulation”, Ph.D. Thesis, ERL, UC Berkeley, June 1990.

[3] S. Borkar *et. al.*, “iWarp: An Integrated Solution to High-Speed Parallel Computing”, *Proceedings of Supercomputing 1988 Conference*, Orlando, Florida.

[4] A. Gunzinger *et. al.*, “Architecture and Realization of a Multi Signalprocessor System”, *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, California, August, 1992.

[5] E. A. Lee, and D. G. Messerschmitt, “Synchronous Data Flow”, *IEEE Proceedings*, Sept. 1987.

[6] G. C. Sih, and E. A. Lee, “Multiprocessor Scheduling to Account for Inter-processor Communication”, Ph.D. Thesis, ERL, UC Berkeley, April 1991.

[7] J. C. Bier *et. al.*, “Gabriel: A Design Environment for DSP”, *IEEE Micro Magazine*, Oct. 1990, Vol. 10.

[8] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems”, Invited paper in the *International Journal of Computer Simulation*, to appear.

[9] M. B. Srivastava, “Rapid Prototyping of Hardware and Software in a Unified Framework”, Ph.D. Thesis, ERL, UC Berkeley, June 1992.

[10] J. Buck, and E. A. Lee, “The Token Flow Model”, *Data Flow Workshop*, Hamilton Island, Australia, May 1992.

FIGURE 3. Photograph of the board