EXECUTION CONTROL AND MEMORY MANAGEMENT OF A DATA FLOW SIGNAL PROCESSOR

Klaus Kronlöf

Helsinki University of Technology
Department of Technical Physics
SF-02150 Espoo 15, Finland

ABSTRACT

The architecture of the Data Flow Signal Processor (DFSP) is discussed with the emphasis on its control mechanism. It is argued that the data flow principle can be efficiently applied to block processing operations of nonrecursive DSP computations, when shared data structures are avoided. Simulation results involving the optimal operand size and the memory use of the control section are presented. Due to the expandability and convenient programmability of the DFSP architecture, the range of its potential applications extends beyond signal processing as demonstrated by a DFSP based database machine.

DATA FLOW CONTROL IN DSP TYPE COMPUTATIONS

The data flow model of computation has been suggested as an attractive alternative to the von Neumann computer architecture. The basic principles are:

(1) Asynchronous execution of operations on the basis of availability of data.

(2) Applicative semantics allowing no side effects to operations.

Programs are represented by data flow graphs, through which data values pass as tokens triggering invocations of operations.

Data flow machines are generally composed of same building blocks: instruction-ready detection, instruction execution, and result distribution. Differences are introduced in the implementation of reentrant programs and data structures. The particular solutions used in the DFSP architecture are strongly impacted by the special properties of DSP computations.

Parallelism in DSP tasks

A real time DSP task involves processing of a continuous stream of incoming signal data to produce a continuous output. The inherent parallelism stems from simultaneously executable operations within a single computation cycle and from the possibility to overlap the computation cycles in time. The latter, i.e. the pipeline parallelism, is the major source of concurrency in many applications.

The degree of the pipeline parallelism in a DSP algorithm is essentially determined by its feedback data dependencies. A recursive algorithm uses, by definition, some results of previous computation cycles as operands. These intercycle dependencies imply a theoretical upper bound to the amount of pipelining. For nonrecursive algorithms, no such limit exists.

High utilization of a typical data flow machine provides for enabled operations enough to saturate its instruction execution pipeline[1]. Overlapped execution of computation cycles is generally required, because the concurrency within a single computation cycle may be insufficient. This means that efficient execution on a data flow machine can be guaranteed only for nonrecursive algorithms. Moreover, the reentrancy of programs is essential for automatic detection of the pipeline parallelism.

Control flow in DSP algorithms

In the majority of DSP algorithms, the flow of computations is independent of data values. The optimal mapping of such algorithms to any parallel architecture can, in principle, be solved prior to program execution. Hence, the runtime scheduling of operations performed by a data flow machine is in many cases redundant. However, the task of mapping an algorithm to a complex architecture is difficult, and no general methods to solve the problem have been presented[3,4].

The overhead due to the redundant data flow control can be reduced by associating complex operations with the actors of the data flow graph. This approach has two benefits: the scheduling effort is significantly reduced, and the a priori knowledge of sequencing can be utilized in the implementation of individual operations[1,2].

© 1983 ACM 0149-7111/83/0600/0230$01.00

Block processing techniques of DSP provide suitable operations for data flow control, e.g. the FFT, FIR filtering of a signal block, crosscorrelation of sample vectors, and deconvolution of a pixel array or a part thereof. Such high level signal processing operations typically are free from side effects, have data value independent control structure, and involve a high computational complexity in terms of elementary arithmetic operations.

Use of data structures in DSP

The principal data structure in real time signal processing is the stream. A stream element may be a single value, an array of signal samples, or a 2-dimensional pixel array. Block processing algorithms can usually be decomposed into high level operations, which have arrays as operands and completely modified arrays as results. Successive cycles of nonrecursive computations are independent, when signal blocks are appropriately overlapped (i.e. the overlap-save technique is used).

The implementation of data structures using a common memory is not suitable for a data flow signal processor. The mechanisms used to allow sharing of structures (e.g. reference counts and linked structures) cause multiple memory references for a single indexed access. The high frequency of accesses typical to DSP computations causes a severe memory contention problem. Furthermore, a complicated memory management system is required.

In DSP computations, severe overhead is not necessarily introduced by allowing tokens to carry structure values. Excessive copying of data is in most cases avoided by proper decomposition of the algorithm. This approach utilizes the high locality of data references characteristic to many DSP computations. Streams can be treated as separate elements carrying color values to indicate the ordering. The storage overhead caused by copying is tolerable, provided there are few data dependencies between computation cycles.

THE DFSP ARCHITECTURE

The following design goals have been taken in order to meet the special requirements of digital signal processing:

(1) Reentrancy of programs is provided by using colored tokens[1]. Each result packet carry a cycle label, which uniquely names of the destination activity template. A new activity template is generated, if no matching template is found.

(2) Data structures are circulated along with operration and result packets. Streams are implemented as separate colored elements. Arrays are conventionally represented using continuous data blocks and indexing. No shared structures are provided.

(3) High level data flow instructions are supported by allowing any number of operands of any type, including structures.

Building blocks of the architecture

The block diagram of the DFSP architecture is shown in Fig. 1. The bank of processing elements (PE) constitutes the execution unit, which performs the signal processing computations. The other parts of the architecture form the control section, which is essentially a data flow instruction execution pipeline.

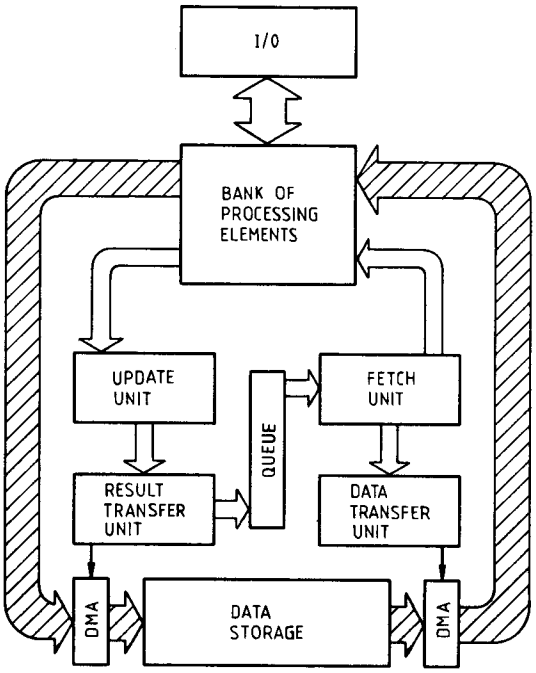

Figure 1. Block diagram of the DFSP.

The double bus architecture of the DFSP separates the data communications physically from the execution control. Signal data is transferred via the shaded buses of the figure. The unshaded buses are used for operation and result packets, which do not contain operand and result values, respectively.

The control section

The distribution of results to destination operations is directed on the basis of the information included in the result packets and the structures of the activity store (not shown in the figure). The update unit performs the matching of operands and allocates the data storage. The result transfer unit controls the transmission of data from the PEs to the data storage. It also detects executable operations. Free PEs are assigned these operations by the fetch unit. The data transfer unit deallocates the data storage upon the transmission of the operands to the executing PE.

Operand matching is done by hashing into the activity store, which is a conventional linear memory. Memory management functions are carried out using the list structures of the activity store. Hashing must be done for each arriving result

packet and memory must be allocated for the first arriving operand of each operation, so considerable effort is associated with the execution mechanism. However, this is not expected to cause a severe bottleneck for the following reasons:

(1) The average rate of incoming result packets is expected to be relatively low due to the high level of data flow instructions.

(2) The operand values are large structures, and they are transferred in parallel with the execution control. Performance degradation occurs only, if the time required for the control exceeds the data transfer time.

Processing elements

The PEs are allowed to be functionally nonidentical in order to capitalize the existing high speed architectures for fixed signal processing algorithms. Frequently used operations (e.g. the FFT) may be executed in dedicated PEs having the appropriate hardware structure.

The I/O-functions take place in special PEs called I/O-processors. This is convenient in signal processing applications, because signal sources and sinks tend to introduce specialized requirements.

A host computer is required to load the application programs of the DFSP. Programs are coded as separate high level operations, which are copied into the local memories of the PEs. After this initialization, the DFSP can operate in stand alone mode.

THE CONTROL MECHANISM

The communication between the control section and the PEs is accomplished using fixed format messages. Initially, all the information about the data flow graph of the application program resides in the local memories of the PEs. Result packets carry the pieces of this information needed by the control section to schedule the operations. A representation of the currently active part of the data flow graph is formed in the activity store, which can be accessed by all the four units of the control section.

Data representations

The essential data objects manipulated by the control section are: result packets, activity templates, and operation packets. Here, they are specified as tuples of information fields.

Result packet. The tokens flowing between actors of the data flow graph are represented by the result packets. Each result packet specifies one token by the following fields:

<OPERATION, CONTEXT, TOTAL_SIZE, RESULT>

The OPERATION field identifies the destination actor of the data flow graph and the CONTEXT field identifies the environment of the invocation. The TOTAL_SIZE field contains the cumulative physical size of operands of the destination operation. This value is used for both the memory management and the detection of executable operations. The RESULT field specifies the position and type of the particular operand represented by the result packet.

Activity template. The specifications of destination operations are placed into fixed format records called the activity templates, which reside in the activity store. The record format has the following fields:

<LINKAGE, OPERATION, CONTEXT,
TOTAL_SIZE, TRIGGER, LOCATION>

The LINKAGE field contains pointers, which build up the structures of the activity store. The TRIGGER and LOCATION fields are used for detecting executable operations and for memory management purposes, respectively.

Operation packet. A portion of the activity template of an executable operation is sent to the executing PE in order to identify the computing task. It has only two fields:

<OPERATION, CONTEXT>

The operation code, which resides in the local memory of the PE, specifies the interconnections of the operation in the data flow graph.

Organization of the activity store

The activity store contains the activity templates of those operations, which have received at least one of the operands, but which are not scheduled for execution. Conceptually, the activity store contains a representation of the active part of the data flow graph.

The organization of the activity store is shown in Fig. 2. The structures support the three major functions of the control section:

(1) Matching of operands.

(2) Management of memory for operand values.

(3) Detection of executable operations.

Operand matching. The associative matching function is based on hashing. A hash index is generated for each arriving result packet using the OPERATION and CONTEXT fields specifying the destination operation. This index refers to a bucket of activity templates.

Buckets are implemented using linked list structures instead of fixed size blocks. The benefits of this organization are the savings in memory space and avoiding special treatment of bucket overflows. The additional list search effort is expected to be tolerable due to short average list lengths.

Memory management. Both the activity store and the data storage are allocated dynamically. The activity store is treated as a pool of fixed size templates. Available templates are marked free and the allocation routine examines templates starting from the most recently allocated. The commonly used list organization for the pool of available records has been rejected due to the feasibility of the circulating allocator in this application.

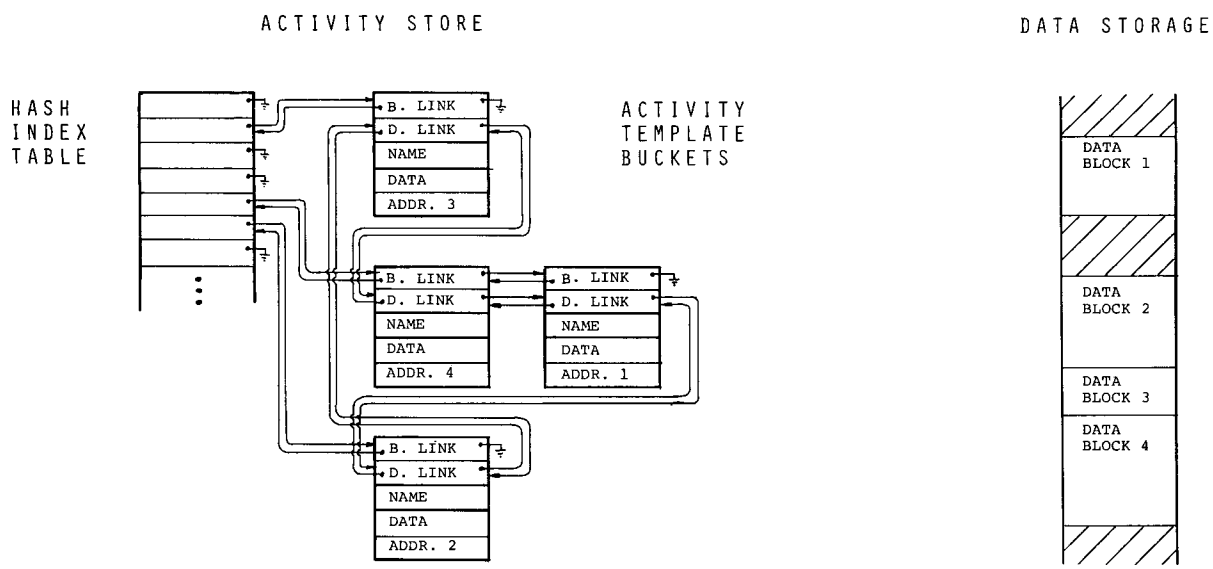

Figure 2. List structures for hash buckets (B) and data storage allocation (D).

A continuos block of data storage is associated with each activity template to store the operand values. All the activity templates are linked together in a circular list, so that the ordering of templates in this list corresponds to the ordering of their data blocks in the data storage. The allocation routine travels around this circular list and uses the first fit principle for selecting the memory block. The starting address of the allocated block is stored in the LOCATION field of the activity template.

Twoway lists are exclusively used in the organization of the activity store in order to aid the memory deallocation. The memory areas of the activity store and the data storage associated with an activity template are made available for reuse by first removing the activity template from both the hash bucket list and the allocation list, and then marking the template free.

The free storage is expected to concentrate after the most recently allocated block due to the uniform and continuous nature of DSP computations. This makes the proposed circulating allocator routines feasible. The DFSP architecture does not include any secondary storage nor does its memory management system recover fragmented storage, because the strict timing constraints of real time DSP prevent from using techniques like virtual memory or garbage collection.

Ready-detection. The TRIGGER field of an activity template is initialized to the value of the TOTAL_SIZE field. After each result transfer, the TRIGGER value is decremented by the physical size of the transferred operand. An operation is enabled for execution, when the resulting TRIGGER value equals to zero.

The detection mechanism allows unlimited number of operands of variable size for each operation. This is considered an essential feature, because the operations are supposed to be complex. The flexibility of operation interfaces supports a functional programming method, where the user defined operations are interconnected by a data flow graph.

Some simulation results

The properties of the DFSP architecture have been investigated by carrying out simulation experiments. The developed discrete event simulator is based on a two-level model of the DFSP. The control section is simulated up to individual accesses to the shared memories, while the PEs are treated at a gross functional level. A more detailed description of the simulation system can be found in [5].

The results given here are obtained from the two examples discussed in [6], i.e. a three sensor tracking task and a real time image processing application. The simulated hardware configuration in both examples has: 64 PEs for I/O and DSP computations; 64 kwords of buffer memory for the data storage; and 1 kword multi-port RAM for the activity store. The control section is assumed to be implemented using commercial microprocessors and other standard components.

Optimal operator granularity. The control section has basically two distinct tasks - the operation scheduling and the data communications. The load distribution among the control units and the data transfer devices is largely determined by the decomposition of the application program into individual operations. Here, the physical size of the result data blocks is used as a measure of the operator granularity.

The update unit has been the major bottleneck of the control mechanism in the simulated microprocessor implementation of the control section. Its average execution time for one result packet has been around 200 µs. However, considerably uniform utilization of the four processors has generally been achieved[5].

For the 4 MHz bandwidth (words) of the data transfer buses, the optimal size of result data blocks has been around 800 words. This value depends linearly on the bandwidth of the buses and inversely on the throughput of the control mechanism. It is also affected by the data flow graph of the application program, because the number of operands widely varies among the scheduled operations.

An average block size below and above the optimum may cause performance degradation due to a bottleneck at the update unit and at the result transfer bus, respectively. Considerably large variation around this average value is allowed without destroying the balanced utilization, because queue modules are used as buffers between the units of the control section (only the template queues are shown in Fig. 1).

The fetch unit is less critical for the performance, because several result packets are generally needed to enable a single operation. The control processors for the DMA devices cause no potential bottlenecks. Moreover, provided the data buses have equal capacity, the overall performance is not limited by operand transfers, because the data transferred as operands to the PEs must first be transferred as results from the PEs.

The optimal level of granularity may be inconveniently coarse for some applications. Considerably finer level is, however, achieved by replacing the standard components of the control section by special hardware. A proposed VLSI implementation is reported in [7].

Memory space requirements. Relatively small memory area is reserved for the activity templates compared to the size of the data storage (the hash index table occupies one quarter of the activity store). The space requirements for the activity store are moderate, because the high level of operations implies small amount of actors in the data flow graph. The wide variations in the data intensiveness of DSP tasks is reflected in considerable differences in the use of the data storage.

The queues have been empty most of the time in the course of the simulations of the examples discussed. No overflows have occurred in spite of the small module sizes (20 cells for the template and result queues, and 10 cells for the data queues). The utilization of the memories in the two examples is displayed in Table 1. The control of nonidentical PEs is accomplished using sets of parallel queues, out of which only the busiest are tabulated.

The first fit allocation strategy for the data storage performs well in the applications discussed. The memory space is kept unfragmented, which gives short execution time for the allocation routine. However, in multitasking applications, where the lifetime of the activity templates varies considerably, fragmentation may cause problems.

PROPOSED APPLICATIONS

The DFSP is developed to meet the specialized requirements of a class of DSP computations. However, the DFSP architecture is potentially applicable to other fields of data processing satisfying the following provisions:

(1) The application task is continuous and a fixed set of programs is used throughout the task.

(2) The use of data structures is highly local and intensive justifying the transfer of whole structures into the PEs.

(3) The application program can conveniently be decomposed into high level operations.

Signal processing

Many signal processing applications introduce the conflicting requirements of high computational capacity and flexibility. The former is generally satisfied by directly mapping the algorithm onto the hardware (e.g. FFT-processors), while the latter implies a programmable architecture.

The processing capabilities of the DFSP can be tailored to meet the requirements of the application, because its architecture does not limit the amount and type of PEs. The shared bus structure establishes a fully general interconnection between the PEs. Thus, new algorithms can be implemented without hardware reconfiguration provided that the total processing capacity is sufficient.

The DFSP architecture provides many advantages for software development. The programmer does not need to bother about synchronisation of parallel activities, because the data flow principle provides implicit synchronisation. This is particularly convenient in multi-input signal applicatios,

	Three sensor tracking task			Real time image processing		
	Min.	Max.	Aver.	Min.	Max.	Aver.
Activity store	13.0	18.8	16.0	2.9	5.8	4.3
Data storage	24.6	36.9	29.7	3.1	7.0	5.4
Template queue	0.0	5.0	0.2	0.0	5.0	0.5
Result queue	0.0	5.0	1.2	0.0	5.0	0.2
Data queue	0.0	10.0	2.0	0.0	10.0	0.4

Table 1. The utilization percentages of the control section memory areas.

such as correlation computations. The architecture also supports structural programming, because programs must be decomposed into modules having well defined interfaces.

Potential applications of the DFSP include transform oriented signal processing (e.g. spectral analysis), image processing, and pattern recognition. Recursive algorithms are generally not efficient for the DFSP, because they do not always decompose into a data flow graph providing a sufficient level of parallelism and operator complexity Therefore FIR-algorithms should be preferred, if filtering is needed as a preprocessing stage in the application.

Database query processing

Processing of relational database queries has many similarities to real time DSP computations. The continuous input stream is operated by fixed programs and pipeline processing can generally be employed. Typical operators of relational algebra queries (e.g. restrict, project, and join) involve all the tuples of the argument relations implying intensive access to data structures. Because of these computational features of database processing, the applicability of the DFSP architecture to a database has been investigated.

Physical database. The database is distributed into several mass storage subsystems. The bank of PEs is divided into classes so that one class is attached to each storage system. This approach utilizes the locality of queries by allowing parallel access to distinct parts of the database.

Representation of relations. Relations are implemented as streams of pages, which are colored according to their ordering. The DFSP architecture supports the page-level operator granularity for actors of the data flow graph. This level is considered appropriate for data flow database machines[8].

Concurrency control and resource allocation. The control of concurrent queries can readily be implemented in the system, because all the operations accessing the physical database are scheduled by a single centralized control section. The control scheme is used both to assure security and integrity for the database and to provide priorities for the users. The PE class concept is extended to a resource allocation scheme, which dedicates distinct subsets of PEs for each query priority class.

Query execution. I/O-processors receive user queries, parse the query tree, and perform the necessary optimization. The produced result packets specify the context (user) of the query. They are destined to the starting operations of the data flow graph, which correspond to the leave nodes of the query tree. Executable operations are sent to those PEs attached to the appropriate mass storage subsystem. The execution of an operation generates a stream of result packets carrying page-level operands. Pipeline processing is applied to successive nodes of the query tree. The result of the query - a message in the case of database modification - is sent back to the I/O-processor performing the query output.

The expandability of the DFSP architecture permits the evolution of the database system to meet changing requirements. The bank of parallel PEs provides also fault tolerance, although the single control section is in series with it in the reliability model. The architecture supports the simultaneous execution of multiple queries from several users by allowing reentrant programs and context identification. Multitasking is essential in a database environment, if system resources are to be efficiently utilized.

In conclusion, the DFSP architecture seems to have attractive properties for relative database processing. However, there are some unsatisfactory features that require further research. The shared buses for the data communications may cause a bottleneck. The locality of queries could be better utilized using a hierarchical bus structure with distributed buffering of intermediate results. In addition, secondary backup storages are required due to the large physical size of relations.

REFERENCES

1. D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn, "A Second Opinion on Data Flow Machines and Languages", Computer, Vol. 15, No. 2, Feb. 1982, pp. 58-69.

2. L.J. Caluwaerts, J. Debacker, and J.A. Peperstraete, "A Data Flow Architecture with a Paged Memory System", Proc. 9th Ann. Symp. Computer Architecture, Austin, Texas, Apr. 26-29 1982, pp. 120-127.

3. J.M. Glass and P. Kotiveeriah, "The Allocation Problem in Distributed Signal Processing", IEEE Trans. Acoust., Speech, Signal Processing, Vol. ASSP-29, No. 4, Aug. 1981, pp. 817-830.

4. K. Kronlöf, I. Hartimo, and O. Simula, "The Compatibility of Computing Algorithms to Parallel Processing Architectures", accepted to be presented at the 1983 IEEE International Symposium on Circuits and Systems.

5. K. Kronlöf, J. Skyttä, O. Simula, and I. Hartimo, "Simulation of a Digital Signal Processing Architecture Based on the Data Flow Principle", Proc. 1982 Int. Symp. Circuits and Systems, Rome, Italy, May 10-12 1982, Vol. 3, pp.1053-1056.

6. K. Kronlöf, J. Skyttä, I. Hartimo, and O. Simula, "Performance of an Experimental Data Flow Architecture for Signal Processing", Proc. 1982 Int. Conf. Acoust., Speech, Signal Processing, Paris, France, May 3-5 1982, Vol. 2, pp. 695-698.

7. K. Kronlöf, I. Hartimo, and O. Simula, "On the VLSI Implementation of a Data Flow Signal Processor", Proc. 1982 IEEE Int. Conf. Circuits and Computers, New York, N.Y., Sep. 28-Oct. 1 1982, pp. 594-597.

8. H. Boral and D.J. DeWitt, "Design Considerations for Data Flow Database Machines", Comp. Sc. Tech. Rep. 369, Univ. of Wisconsin-Madison, Oct. 1979.