MagicEight

An Architecture for Media Processing and an Implementation

John A. Watlington

MIT Media Laboratory

December 13, 1999

Abstract

This thesis proposes a mechanism, streams, for overcoming many of the problems associated with the parallel processing of media. A programming model and runtime system using the stream mechanism, MagicEight, is also proposed. MagicEight supports medium to coarse grain parallelism, using a hybrid dataflow model of execution. Multidimensional streams of data elements provide a scalable means of both obtaining parallelism and ameliorating an indeterminate memory access latency. The machine architectures which MagicEight is intended to support range from a single general purpose processor to heterogenous multiprocessor systems of two to two hundred processors interconnected by communications channels of varying capabilities. In the multiprocessor case, some of the processors may be specialized — capable of executing a restricted set of algorithms much more efficiently than a general purpose processor.

1 Introduction

While the ability of personal computers to acquire, process, and present video and sound has now been established, the computational requirements of many media applications exceed that provided by a single general purpose processor. My thesis is that streams are a mechanism for enabling efficient dynamic parallelization of the computational tasks typically found in media processing. I am also proposing a programming model for media processing using this mechanism. The model is a variant of hybrid dataflow, utilizing multidimensional streams as both a basic data type and a mechanism for synchronization and obtaining parallelism. It supports machine architectures containing a heterogenous mix of processors.

In order to provide higher compression, greater flexibility, and more semantic description of scene content, video[1] is increasingly moving toward representations in which the data are segmented not into arbitrary fixed and regular patterns, but rather into objects or regions determined by scene-understanding algorithms [18][23][30][7][6]. These structured (or object-based) representations are effectively sets of objects and "scripts" describing how to render output images from the objects. The media being presented is generated at the receiver, not merely decoded, allowing the presentation to adapt to receiver capabilities, viewing situation, and user preferences.

Custom processors operating in parallel and using hardwired communications networks are capable of meeting the computational demands of media processing for a given algorithm/application, yet the flexibility needed to support different algorithms (and thus object-

[1] While this thesis is applicable to a range of media, including audio, the computational requirements of video are the primary motivation (see section 2).

based media) is difficult to provide with these architectures. Single "general-purpose" processors, now often with specialized instructions/datapaths for manipulating small data elements in parallel (*e.g.* using a 32-bit ALU to process 8-bit R,G,B pixel values simultaneously [28][20]), provide adequate flexibility and are showing promise of meeting the needs of the current generation of media applications. Yet algorithms and applications which require tens to thousands of times more computation and memory bandwidth than current applications are being developed. Programmable parallel architectures will remain attractive for all but the cheapest or most limited media applications, since these requirements of media processing greatly exceed the capabilities of single processors.

Irrespective of the coding method, computational needs for video are likely to increase greatly in coming years. Digital video, unlike digital audio, is far from operating at human perceptual limits. As display technologies and communications bandwidth permit, higher definition systems will add to the computational demands. Alternative output technologies, for example the holographic video displays developed at the MIT Media Laboratory [27][32], push these demands still further.

The programming model described in this thesis is an attempt to support computationally demanding media tasks in an environment in which the programmer can take advantage of parallelism and specify real-time performance without needing to know details of the hardware architecture(s) used to execute the tasks. Thus differently scaled or architected systems should be able to execute the same application software, and specialized processors may be utilized without the explicit cooperation of the application developer. I am proposing systems that utilize whatever resources are available currently in the local area — the system attempts to dynamically execute an application on any processing nodes found idle nearby. Consider, for example, a VRML viewer on a personal computer borrowing cycles from the rendering engine of a video game in the next room, or several PCs working together to achieve real-time media encoding.

This model inherits from dataflow the attributes of programmer transparent parallelism and resource sharing. It also allows the efficiency of the static pipeline model — processing units may be directly coupled to minimize (off-chip) memory accesses. It extends current hybrid dataflow[2] architectures by supporting data primitives which are multidimensional streams of scalars or vectors. These are partitioned at runtime to provide an appropriate scheduling granularity[3], minimizing the overhead associated with each partition.

In order to evaluate this thesis, I am proposing to develop a software system for testing this programming model. This implementation, MagicEight, will utilize traditional workstations interconnected by a local area network and serve several purposes. It will allow research into the resource requirements (amount of processing overhead induced by the stream mechanism, stream buffer size, etc.) of such a system. It will also allow the development and testing of applications without the specialized hardware (or multiple processors) needed for real-time performance. While the software being written is intended to ultimately support specialized processors and native implementations (where MagicEight is the operating system), these are not being included in the scope of the proposed dissertation research.

In order to test the system appropriately, I am planning to implement a video segmentation

[2]Hybrid dataflow uses dataflow techniques to schedule sequences of instructions. Pure, or fine-grained, dataflow schedules individual instructions.

[3]The number of instructions being scheduled at one time is referred to as the scheduling granularity. Fine-grain implies that one or two instructions are being scheduled. Medium granularity, targetted by this thesis, implies a scheduling quanta of between ten and a thousand instructions.

application based on the multimodal segmentation research done within our group [11]. This application uses both a large number of low-level vision operations and medium-level operations which track the objects and classify pixels to corresponding objects. This is an algorithm which currently runs sixty to one thousand times slower than real time on our fastest Alpha workstation, working with small (320x240) images, providing ample complexity for testing the MagicEight model.

The stream mechanism and MagicEight programming model are proposed by this thesis in order to solve a computational problem posed by processing media (in particular, video). They were designed to take advantage of the characteristics of media processing (the large amount of data and the data independence of typical media data access patterns), and rely on them for efficient execution. While it is hoped that the programming model performs well enough for effective general purpose computation, it is predicted that the data parallelism provided by media processing (and similar applications) will be necessary for a significant performance gain.

Section 2 describes the problems addressed by this thesis in more detail, along with previous work addressing them. A better description of the thesis may be found in Section 3, along with a description of the example implementation (MagicEight). Finally, a rough estimate of the resources required and an estimated timeline is provided in Section 4.

2 Problem Description

In simple terms, the problems faced by a media processing system are twofold: how to build a machine with the required performance, and how to efficiently program it once built.

It is tempting to liken the problems associated with digital video today to those faced by early forays into digital audio. Early attempts at processing digital audio required the use of specialized architectures and processors; most audio processing algorithms can be implemented today using a single chip programmable digital signal processor, with many algorithms implementable on today's general purpose microprocessors. Until now, video processing systems have used specialized architectures and processors. In the past few years several implementations of programmable digital video signal processors have appeared, especially targeted at implementing low resolution motion compensated transform based coding. The data intensive nature of the image processing task, however, differentiates it from audio processing. While the input/output data rates in a typical digital audio system are determined by the sensitivity of the human auditory system, the video data rates will be technology limited for many years to come. Six channels of uncompressed audio data require only 4.2 Mbits/sec of data[4], but an uncompressed video channel today requires between 66 and 26,000 Mbits/sec of data[5], making parallel processing necessary for most applications.

2.1 Object-Based Media

The algorithms and computational requirements of model-based media are still the subject of research, but it seems reasonable to predict that no one image representation or compression scheme will emerge as a clear winner. Instead support for multiple representations is likely to be necessary [6][21][22]. These multiple representations may be used for different applications: a full length feature movie is likely to use a different representation than an interactive video

[4]Computed using a 44 KHz sampling rate and 16 bits per sample. While these parameters may be contested, they are correct within a factor of 4.

[5]Lower data rate is NTSC broadcast video. Upper data rate is 6000 samples by 2000 lines at 72 frames per second using three color components each represented by 10 bits.

game. They will also be used within a single application: an interactive video game might incorporate objects represented graphically in 3D over background video sequences encoded using MPEG.

This multitude of media representations prevents the traditional solution: specialized processors operating in parallel with a fixed interconnect; and instead suggests that a more flexible approach must be developed. Frequently, a "completely" programmable solution is proposed. Unfortunately, the processing power of the most powerful single processor tends to be insufficient for typical object-based video processing task. Thus we are forced to consider programmable parallel solutions, possibly with some of the processors specialized for a class of algorithms.

2.2 Obtaining Parallelism

A traditional computer program (regardless of the high level language used) is written in an imperative manner. One or more sequences of instructions, with occasional conditional branches, are defined by the programmer and executed sequentially by a single processor. In such a program specification, it is difficult to separate the dependencies (limits on parallelism) introduced by the programmer from those intrinsic to the algorithm.

Still, a small amount of parallelism (2 – 4x) is typically obtained in a manner transparent to the programmer, by scheduling adjacent instructions in the instruction sequence to execute in parallel. This may be done either statically at compile time (a Very Long Instruction Word, or VLIW, approach) or dynamically at runtime ("superscalar").

Parallelism may also be obtained at other levels through any of several mechanisms: parallel threads, or processes; usually targetting a particular computer system architecture and granularity of parallelism (typically coarse-grained). The partitioning of an algorithm into a relatively fixed number of separate threads must currently be done before compile-time by the application programmers.

While a fine grained dataflow execution model would theoretically expose the highest amount of parallelism in an application, the overhead of synchronizing every instruction has been proven costly. A compromise seemingly well suited for today's processors is hybrid dataflow [16], where the dataflow element used for synchronization and scheduling represent short sequences of instructions which only access global variables at the start and end of the sequence. The instruction sequences are then scheduled and executed in a dataflow manner, amortizing the cost of synchronizing the sequence by the number of instructions in the sequence.

2.3 Memory Latency

Based on experience with current prototype systems, a system for processing structured video requires on the order of 3 to 20 frames of memory. A single device targeted for broadcast resolution will require from 20 to 150 Mbits of memory, precluding the sole use of on-chip memory. Even if wafer scale integration or multi-chip modules are used, the device memory would be physically separate from the processor, presenting a large access latency. This is a serious problem in any processing system — the speed of a processor is irrelevant if it is stalled waiting for data. This latency problem is greatly exacerbated by parallel processing, where it is more likely that the data being read is located remotely from the processor, and accesses may conflict with those made by other processors.

The typical approach to reducing latency is the use of a cache — a relatively small amount of lower latency memory containing a copy of sections of the slower memory that the processor is using or likely to use. Although this approach can be very effective in single processor systems, given typical application memory access patterns its performance on media processing is less than optimal. There are two reasons for this: the large amount of data being processed, and atypical data access patterns. The amount of data typically accessed by a media algorithm typically exceeds the size of a typical cache, diminishing the likelihood that desired data will be found in the cache[6]. But a cache improves performance not only through maintaining a local copy, but also by prefetching data in addresses linearly adjacent to the requested one. Unfortunately, this automatic prefetching can degrade performance when accessing data sparsely (with a step between samples greater than the size of a cache line).

When using the cache mechanism in a multiprocessor system, great care must be exercised to ensure the validity of the data in the cache and the memory. The overhead of maintaining coherence does not scale well, requiring either that all processors monitor all memory accesses, or that lists of data copies be maintained and used. Cache coherence schemes have been shown useful in systems of 2 – 4 processors [1].

2.4 Specialized Processors

Within a given technology, it is always easier to achieve a specific performance if the computational task is completely specified *a priori*. A specialized processor may be optimized for a single task, or made flexible enough to perform a range of tasks. While it is difficult to obtain the performance needed for video processing from a single general purpose processor, it is quite

[6]The cache overflow problem tends to be an artifact of the particular algorithm implementation. One of the goals of this thesis is to show that streams may be used to partition algorithm execution (at runtime) in such a way to avoid cache overflow.

possible to build systems using specialized processors with the required power. Object-based media requires additional flexibility, but there are often similarities between different media representations, both in data types and common computational operations, which may be utilized. For example, both a subband representation of video and a block-transformed one (M/JPEG) utilize an inner loop of multiplication and addition which maps well onto small SIMD arrays of fixed point multiply-accumulate units.

Media processing systems currently utilize a broad spectrum of parallel processor architectures, from the relatively inflexible fixed pipelines of specialized processors common in "3D graphics accelerators" and compressed motion image sequence decoders to highly flexible MIMD arrays of general purpose microprocessors. Given the magnitude of the computational task, it is not surprising that processors specialized for a particular media processing task are popular. In previous years, the high cost of silicon chip area precluded incorporating specialized processors into a design intended for general purpose computing. This is changing, as even mainstream microprocessors have adopted architectural features specialized for media processing [28][20]. Further impetus is provided by the development of dynamically reconfigurable logic [4]. The difficulty lies not in building specialized processors, nor in the silicon area they require, but in providing an efficient, general, programming model for them.

3 My Thesis

My thesis is that a stream mechanism may be used to ameliorate the problems associated with the parallel processing of media. Streams provide a scalable means of obtaining parallelism, support specialized processors, and allow the indeterminate latency of memory accesses to be ignored. In order to fully exploit the capabilities of streams, a programming model utilizing them is also being developed [31].

The goals of the MagicEight programming model are:

- to allow efficient use of a set of heterogenous processors, interconnected through channels of varying capacity
- to support specialized (*esp.* non-Von Neuman) processors
- to allow a program to efficiently execute on systems with varying numbers of processing nodes, within a range of 1 to approx. 200, in a programmer transparent manner

3.1 The MagicEight Programming Model

I begin by suggesting that applications be written in an intensional manner, where algorithms are represented using a minimal amount of programmer imposed dependencies. In effect, writing in an intensional language is expressing only the basic equations (or intensions) relating input, intermediate, and output data variables, leaving the details of how it is executed to the interpreting system and maximizing the functional parallelism available to the system. Another example of a multidimensional intensional programming language is Lucid[3].

All variables (non-constant data values) are viewed as streams, where a value at a particular stream location (or context) may only be assigned once. Variables that change over time are thus explictly identified as such. When specifying the stream coordinates over which a task is to be evaluated, it is possible to specify an endpoint along a dimension (either the oldest known value or the newest value, if the dimension is time). It is also possible to specify infinite ranges — a function should be applied to all values along a dimension (typically some time dimension).

Instead of implementing a fine-grained dataflow system to execute an application written in such a manner, I instead propose the use of a hybrid dataflow execution model. In such a model, small sequences of instructions are scheduled and executed in a dataflow manner. This thesis extends hybrid dataflow to include dynamic data parallelism, scheduling the evaluation of a set of output values (the *stream context*) instead of individual values. This extension allows the *pipelining* of function applications, either in software or hardware.

If one views hybrid dataflow as being the scheduling of a function (and its parameters) instead of individual instructions, the MagicEight programming model schedules functions and a number of different parameters the function should be called with, thus amortizing the overhead of scheduling even further. The size of the stream contexts (the number of function applications) used for scheduling is determined dynamically by the number of processing nodes available at any one time. This allows efficient scheduling on single processor machines, yet can take advantage of multiple nodes through increased data and functional parallelism.

A MagicEight user program will thus contain two different means of algorithm specification: dataflow and imperative. The dataflow portion may be specified using a graphical programming language, or simply a set of data structures defined using an imperative initialization routine (the way Cheops [8] was usually programmed). It is hoped, however, that dataflow portions of MagicEight applications will be written in an intensional language, such as Lucid [3]. The imperative portions are written using any traditional programming language (such as C).

The programming model includes a runtime system which interprets a program representation, in addition to providing the features normally associated with an operating system. This *resource manager* allows an algorithm to be executed on a variety of actual systems by dynamically mapping the algorithm onto the available number and type of processing units. It allows applications executing concurrently to share a limited pool of processor, memory, and communication resources, providing graceful degradation of a system (fair access to resources) when overloaded. It also performs the run-time partitioning of streams to exploit data parallelism.

3.2 MagicEight Program Elements

The *task token* is the basic unit of execution in the MagicEight architecture. It contains the tag of the *function* being called, the data the function is being called to process, and where the output goes. Non-constant input parameters of the task and its output are represented by the tag of a stream and the appropriate context (multidimensional volume) of the stream to process.

The function referenced by each task token is an object representing an "instruction sequence" of an algorithm, with multiple machine implementations. It defines the input and output parameters that a particular function expects, the stream parameter data access patterns used, and information about different implementations of the function. The implementation-specific information includes which machine it runs on, statistics about how fast it runs on that machine, and the tag of the code object with the actual "instructions" for that machine. The information contained in the function object is used to properly schedule a task applying this function.

The code objects are typically specified in an imperative manner such as C — to take advantage of modern compilers (especially on VLIW processors) — and are pre-compiled into extended cross-platform libraries. Specialized processors are supported as well by these libraries. An interface routine is provided for each accelerated function which configures the specialized processor as needed, then takes the function parameters and writes them to the specialized processor. The reconfiguration time for a specialized processor is considered by the scheduling algorithm, as it may be considerable in some cases (*e.g.* FPGAs).

3.3 Stream Mechanism

The notion of a "stream" of data has pervaded computing since at least the 1960s [7]. The search for programming languages that effectively express the parallelism inherent in an algorithm has also driven the development of a stream mechanism [12][2][19]. The stream definition typically used is an ordered sequence of scalar values. Although streams of compound objects have sometimes been supported (in particular, a stream of streams), these streams are in general one-dimensional.

Probably the most successful example — the UNIX operating system — uses one-dimensional character streams at both the user and programmer interfaces. The ANSI C standard programming model derived from it defines only a stream interface for data I/O. The reason given for this decision is that, for performance reasons, most programs that use a raw I/O interface implement internally the equivalent of a stream interface (with its implied buffering)[25].

Previous stream mechanisms have commonly been utilized by a system implementation in one of two distinct manners: in many cases they are used only for purposes of algorithm

[7] Knuth describes the history of a precursor to streams, coroutines, in his compendium: *The Art of Computer Programming* [17].

specification[12][3][13]. Once the algorithm is translated for execution, the data is treated as an ordered sequence of individual data objects. In this dataflow model, "tokens" consisting of single instructions along with a reference to data values to which they should be applied are generated for each array element in the stream. Another approach, supporting a stream primitive data object, allows efficiencies in data storage not possible in the pure dataflow model [10][24][15][5]. The former approach provides the maximal parallelism, but incurs a high scheduling overhead and is usually proposed for execution only on fine-grained dataflow architectures. The latter approach is amenable to a coarser granularity of processing, which allows the amortization of the scheduling overhead over a number of stream data elements.

Streams have also been proposed before as a means of overcoming memory latency [14]. If a memory access pattern is predetermined, ie. not dependent on the data being processed, merely on the algorithm, then the processor performance may be made independent of the memory access latency by proper data pre-fetching. Hardware prefetching in general only supports simple (linear) access patterns, whereas software prefetching incurs additional instruction stream bandwidth [9]. In either case, the explicit identification of a function's multidimensional data access pattern provided by the stream mechanism should allow more accurate data prefetching.

In MagicEight I propose the novel mechanism of using multidimensional streams as a primitive data type supported by the execution model and the operating system. It differs from previous examples in that it defines the intensional mapping between the multidimensional array and the one-dimensional stream sequence (the access pattern) to be used at the source and destination of each stream [29]. The properties of those mappings are then used to determine implementation requirements such as minimum buffer sizes and available data parallelism.

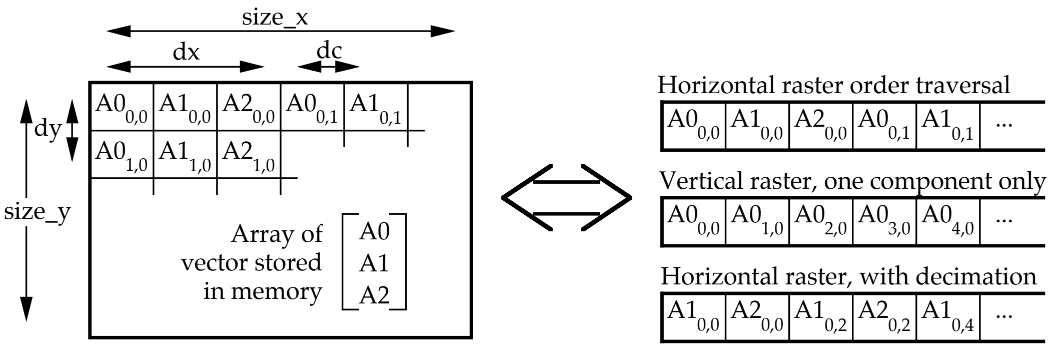

Figure 1: A vector array in memory and streams generated from it

A stream is read or written by traversing through a multidimensional array of vectors in memory in a regular manner. Figure 1 illustrates a two-dimensional array of three-dimensional vectors (which could represent a color image), and three stream generated from it using a variety of simple access patterns.

Instead of devoting a large amount of silicon area to cache memory, and attempting to predict the memory access pattern at runtime, MagicEight proposes decoupling memory accesses from data processing using the stream mechanism. Most image processing data access patterns are pre-determined (data independent), allowing the separation of data address generation and fetch from processing. This provides a processing throughput that is independent of the latency of remote memory accesses. The amount of hardware support for streams may vary from machine to machine within a system. The "simplest" support for streams is an explicit (software) cache prefetch/clear mechanism on a general purpose processor. More complex support includes separate address generators capable of transferring data to/from specialized processors independently [8][26].

3.4 Eduction

There are two basic methods of evaluating the dependency graph which describes a program: demand-driven (or call-by-need) and data-driven (call-by-value). A data-driven approach performs a computation as soon as the required input values are available. While this ensures that a computation is performed as soon as possible, it generally results in unneeded computations being performed. Visualize, for example, a program that only wants to output a small region of interest of a large input dataset.

The demand-driven evaluation, or *eduction*, of a dependency graph is described by two rules:

1. The need for a data value at the output of a process causes it to be demanded.

2. If a data object (or a particular sub-context of one) is demanded, then and only then are values demanded that are known directly to determine the data object.

Thus, eduction is simply tagged, demand-driven dataflow, where the "tag" includes the multidimensional context of the data being processed [3].

When particular contexts of a set of streams is demanded, the streams are partitioned across the nodes of the systems in a cost constrained manner. Even in a single processor system, streams may be partitioned in order to maximize the use of cache memory. This partitioning is done in a distributed manner. Once partitioned and assigned to a processing node, a task is held until all needed parameter and function data is available in storage local to the node. It is then moved into a ready queue, from which it is executed when the processor becomes available. In order to support real-time systems, tasks may be educed with a desired time of execution, which is used to prioritize the tasks in a processing element's ready queue.

4 Resources and Timeline

4.1 Deliverables

In order to defend my thesis, I intend on providing experimental results comparing the MagicEight programming model to a conventional one, and testing its performance across a range of systems. In order to quantify the overhead introduced by the MagicEight system, I propose showing the performance of a single processor system relative to a similar, conventionally programmed application (using relatively simple applications).

I also intend to provide results showing the performance of several test applications across a variety of system configurations. This will provide information about the scalability of the programming model, as well as the performance of the partitioning algorithms. In particular, the task partitioning proposed by the system may be compared to published "optimal" results.

The most complex test application I am contemplating is an implementation of multi-modal segmentation [11] with the goal of accurately tracking and segmenting people and other objects out of motion image sequences in real-time. Current implementations of this algorithm run 60 to 1000 times slower than real time on our fastest Alpha workstation, working with small (320x240) images. Accelerating this application will provide ample complexity for testing the MagicEight model, as well as enabling object-based applications that require accurate segmentation of people and other objects from complex scenes.

An implementation of the MagicEight system, along with ancillary tools, will be written in order to perform the above tests. This implementation is primarily intended for use with internetworked workstations, on top of a native OS. Supported platforms will be Compaq Alpha (running Digital Unix), Hewlett Packard HPPA (running HPUX), and Intel x86 and Motorola PPC (running Linux). If suitable specialized processors become available, their support will be attempted.

I'm not proposing to build a system that executes every program it is possible to write, just those programs which contain computational structures found in typical media applications. There are limitations to stream based programming — discovering them is one reason to build such a system. Likewise, I'm not proposing to provide a pretty user interface to the intermediate data structures used to represent a MagicEight program.

The basic infrastructure of the MagicEight system software has already been written and tested in a heterogenous environment. The programmer's interface (between MagicEight programs and the system) has been defined, as have the internal data structures used to represent system element. The remaining work is that directly related to this thesis: developing and testing stream partitioning and scheduling algorithms for media tasks, and developing the test applications.

4.2 Resources

The resources required for this project are: software tools for building the MagicEight system; sufficient internetworked machines to test the scalability of the system; and I/O devices for the complex test application. The software tools (compilers, assemblers, linkers, and debuggers) I am using are from the Free Software Foundation, as well as "vendor provided tools" on some platforms. These are already available.

I am proposing a system that supports scalabilities from one to two hundred processing nodes. The majority of the testing, however, can take place on systems of one to eight processors. The Compaq Alphas currently around the garden area will suffice. For the occasional larger test, additional computers on student's desks will be used resulting in a more heterogeneous mix. Testing on a system larger than 64 nodes will probably not be performed.

The I/O devices will likely consists of a video camera, a digitizing card, and either a X display or an NTSC one. All of these devices are already available within the group.

4.3 Timeline

Date	Milestone	
	Research	Other
Feb. 1999	Scheduling simple apps	Proposal accepted (?)
March	Runtime linking	Language Definition
April 16		Ian's first birthday
May	I/O support	
July	Complex task developed	Diss. first draft
September	Complex scheduling	
December		Dissertation completed
Jan. 2000		Thesis Defense

Table 1: Timeline

References

[1] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Pubs., Boston, 1988.

[2] S.J. Allan and R.R. Oldehoeft. A stream definition for von neumann multiprocessors. In *Proc. 1983 Int'l Conference on Parallel Processing*, pages 303–306, Aug 1983.

[3] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming.* Oxford Univ. Press, New York, 1995.

[4] Peter M. Athanas and A. Lynn Abbott. Real-time image processing on a custom computing platform. *IEEE Computer*, pages 16–24, Feb 1995.

[5] Shuvra S. Bhattacharyya and Edward A. Lee. Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms. *IEEE Trans. on Signal Processing*, 42(5), May 1994.

[6] V. Michael Bove, Jr. Multimedia based on object models: Some whys and hows. *IBM Systems Journal*, 35(3 & 4), 1996.

[7] V. Michael Bove, Jr., Brett D. Granger, and John A. Watlington. Real-Time Decoding and Display of Structured Video. In *Proc. IEEE Int'l. Conf. on Multimedia Computing and Systems '94*, Boston, MA, May 1994.

[8] V. Michael Bove, Jr. and John A. Watlington. Cheops: A Reconfigurable Data-Flow System for Video Processing. *IEEE Trans. on Circuits and Systems for Video Processing*, 5(2):140–149, April 1995.

[9] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proc. Fourth Int'l Conf. on Architecture Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991. ACM. QA76.9.A73.S95 1991 in Barker Lib.

[10] L. J. Caluwaerts, J. Debacker, and J. A. Peperstraete. Implementing streams on a data flow computer system with paged memory. In *Proc. 10th Annual Int'l Symposium on Computer Architecture*, pages 76–83, 1983.

[11] Edmond Chalom. *Statistical Image Sequence Segmentation Using Multidimensional Attributes.* PhD thesis, Massachusetts Institute of Technology, January 1998.

[12] J. B. Dennis and K. S. Weng. An abstract implementation for concurrent computation with streams. In *Proc. 1979 Conf. on Parallel Processing*, pages 35–45, Aug 1979.

[13] J. L. Gaudiot. Methods for handling structures in dataflow systems. In *Proc. 12th Annual Int'l Symp. on Computer Architecture*, pages 352–358. ACM, 1985.

[14] James R. Goodman, Jian tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P. B. Schechter, and Honesty C. Young. Pipe: A vlsi decoupled architecture. In *Proc. 12th Annual Int'l Symposium on Computer Architecture*, pages 20–27, 1985.

[15] R. W. Hartenstein, A. G. Hirschbiel, and M. Riedmüller. A novel asic design approach based on a new machine paradigm. *IEEE Journal of Solid State Circuits*, 26(7):975–989, July 1991.

[16] R.A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th Annual Int'l Symposium on Computer Architecture*, pages 131–140. ACM, 1988.

[17] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA, 1973.

[18] Murat Kunt, A. Ikonomopoulos, and M. Kocher. Second-Generation Image-Coding Techniques. *Proceedings of the IEEE*, 73(4):549–574, 1985.

[19] Edward Ashford Lee. Representing and Exploiting Data Parallelism using Multidimensional Dataflow Diagram. In *Proc. IEEE 1993 Intl. Conf. on Acoustics, Speech and Signal Processing*, pages I–453 – I–456, April 1993.

[20] R. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[21] MPEG-4 SNHC verification model 3.0. http://drogo.cselt.stet.it/mpeg/index.htm.

[22] MPEG 7. http://drogo.cselt.stet.it/mpeg/mpeg-7.htm.

[23] Hans Georg Musmann, Michael Hotter, and J. Ostermann. Object-Oriented Analysis-Synthesis Coding of Moving Images. *Signal Processing: Image Communication*, 1:117–138, 1989.

[24] J. S. Onanian. A Signal Processing Language for Coarse Grain Dataflow Multiprocessors. Master's thesis, Massachusetts Institute of Technology, June 1989.

[25] P. J. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[26] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proc. of the 31st International Symposium on Microarchitecture*, December 1998.

[27] Pierre St.-Hilaire, Steven A. Benton, and Mark Lucente. Synthetic aperture holography: a novel approach to three dimensional displays. *Journal of the Optical Society of America A*, 9(11):1969–1977, Nov. 1992.

[28] M. Tremblay, J. M. O'Connor, V. Narayanan, and H. Liang. VIS speeds new media processing. *IEEE Micro*, 16(4):51–59, August 1996.

[29] John Watlington and V. Michael Bove, Jr. Stream-based computing and future television. In *Proc. 137th SMPTE Technical Conference*, Sep 1995.

[30] John A. Watlington. Synthetic Movies. Master's thesis, Massachusetts Institute of Technology, May 1989.

[31] John A. Watlington and V. Michael Bove, Jr. A system for parallel media processing. In *Proc. of the Workshop on Parallel Processing in Multimedia*, Geneva, April 1997.

[32] John A. Watlington, Mark Lucente, Carlton J. Sparrell, V. Michael Bove, Jr., and Ichiro Tamitani. A hardware architecture for rapid generation of electro-holographic fringe patterns. In *SPIE Proc. #2406-23 - Practical Holography IX*, Bellingham, WA, Feb. 1995. SPIE.