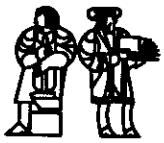
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Summary of 1987 Dataflow Workshop in Japan

Computation Structures Group Memo 282
January 1988

Akihiko Konagaya, Translator

On study leave from NEC Corporation at the MIT Laboratory for Computer Science, 1987-88.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Summary of 1987 Dataflow Workshop in Japan

Akihiko Konagaya*

Computer System Research Laboratory
C&C Systems Research Laboratories
NEC Corporation
1-1 Miyazaki 4-choume, Miyamae-ku
Kawasaki, Kanagawa 213, Japan

ARPA: konagaya@xx.mit.edu
UUCP: konagaya%cslv4.nec.junet@uunet.uu.net

January 22, 1988

It is often said that there is little innovative work done in Japan. However, after reading this paper, you will discover that the lack of information about innovative work is the result of there being very few Japanese research papers translated into English.

This paper is the compilation of all the papers presented at the dataflow workshop held in Japan on October 21-23, 1987. Unfortunately, all but two of the papers were written in Japanese. The workshop was rather informal but the quality is comparable to that of international workshops. In addition, the workshop covers most of the parallel computer research activity now going on in Japan, in spite of the title "dataflow".

The workshop consisted of three special lectures, eleven sessions and one panel discussion (not included in the proceedings nor in this paper). The following lists the sessions and the number of papers in each sessions.

Special lectures		(3)
Session 1A	Language	(4)
Session 1B	Control Methodology	(4)

*On a study leave in the MIT Laboratory for Computer Science

Session 2	Evaluation	(3)
Session 3	Architecture(1)	(3)
Session 4A	Macro Dataflow	(3)
Session 4B	Massive Parallel Machine	(3)
Session 5A	Special Machine and Network	(3)
Session 5B	Structural Data	(3)
Session 6	Architecture(2)	(3)
Session 7	Software Environment	(3)
Session 8	Data-driven Processor	(4)

As reflected in the workshop, Japanese parallel computer research projects can be divided into four groups. The first group includes classical dataflow people at NTT[5][16][33], ETL[19][27][30], Oki[14]., Gunma University[11] and Keio University[8]. They have come up with new ideas for combining dataflow and control flow in order to achieve general purpose parallel computers. Their motivations are based on their research experiences on the prototype dataflow machines developed in the past five years.

The second group consists of newcomers to dataflow at NTT[25], Tokyo University[24], Osaka University[6-7][9][36-39] and Astec Inc.[10]. Some of these people come from a switching system background rather than a computer system background. From their points of view, computation is just a part of communication. Such view points might result in a break-through for parallel computation.

The third group consists of conservative but practical parallel architecture projects in several industries and one institution. NEC reports on a commercial dataflow machine, NEDIPS[31][34]. Hitachi reports on simulation results of a macro dataflow machine[20], and Toshiba reports on the design philosophy of an AI-oriented massive parallel processing system[21]. ICOT reports simulation results of a parallel inference machine[13][28][29]. These reports show that Japanese computer industries are ready to start parallel computer system development.

The fourth group consists of innovative parallel architecture projects in universities. Iwate University proposes a unique processing element based on a "processing in memory" concept[23]. Tohoku University proposes a MISD-type parallel computer for FP[15]. Tokushima University reports a binary tree computer for neural network processing[22]. Hiroshima University proposes a XY-double tree architecture[32]. Tokyo University proposes a parallel high-level language-oriented machine[18]. Their architectures may be premature in some ways, but worth considering.

The workshop also contains parallel language proposals[4][35], theoretical observations for dataflow computation[12], self-routing multi-stage networks[26], and parallel processing applications[1][2][3][17].

I thank Prof. Arvind for giving me the opportunity to publicize this workshop and Natalie Tarbet for her valuable comments in regard to this paper. I hope that this paper will be useful to those who are interested in parallel computation, and that more people will be interested in Japanese research activities and appreciate them.

Special Lectures

[1] Matsumoto,Yuji (ICOT): On Parallel Parsing, Dataflow Workshop 1987, pp. 1-7.

Matsumoto discusses the latest topics in parallel natural language parsers; a logic programming based model and two connectionist models. The former is based on his work at ICOT. He demonstrates how naturally GHC (Guarded Horn Clause, parallel logic programming language developed at ICOT) can express the parallel parser. Two connectionist models, the Fanty and Waltz & Pollack models, are introduced for comparison to the logic programming model.

[2] Ishida,Toru (NTT): Parallel Execution of Production Systems, Dataflow Workshop 1987, pp.9-16.

Ishida proposes a new parallel production system in which multiple production rules are fired in parallel. The simulation results show that 4-8 rules can be applied in parallel by analyzing the data dependency of production rules.

[3] Kitsuregawa, Masaru (Tokyo University) Parallelism in Database Processing, Dataflow Workshop 1987, pp.17-24.

A good survey of parallel database machines is reported. The paper presents various search models, architectures including old models, such as RAP (Tront University), CASSM (Florida University) and CAFS (ICL), and the latest models such as Relacs (Syracuse University), systolic array (CMU), the connection machine (Thinking Machine, Inc.), DIRECT (Wisconsin Uviv.), Grace (Tokyo University), DELTA (ICOT), GAMMA(Wisconsin), Teradata DBC/1012, IDM(Britton Lee, Inc.), as well as a function disk system (the author's). Kitsuregawa also discusses the simulation results of his machine on the Balance 21000.

Session 1A Language

[4] Kono,Shinji and Tanaka,Hidehiko (Tokyo University) : Data-driven Object-Oriented Language DinnerBell; A Bridge Between Object-Oriented Language and Fine-Grain Parallel Execution, Dataflow Workshop 1987, pp.25-32.

The authors propose an interesting fine-grain parallel language based on the Actor model. The point is in its use of synchronization mechanism of message passing from more than two objects; they call it "join". The language is also a kind of a dataflow language in the sense that it adopts a single-assignment scheme and a message-driven object invocation scheme.

[5] Ogawa,Mizuhito and Ono,Satoshi (NTT): Anomaly Detection of Functional Programs Based on Global Dataflow Analysis, Dataflow Workshop 1987, pp.33-40.

The authors propose an anomaly detection algorithm for a functional language with lazy-evaluation. The algorithm can detect redundant function arguments and anomalous function definitions at the inter-function level, as well as redundant local variables and undefined local variables at the function level. To deal with nonstrict functions in the anomaly detection algorithm, the authors provide evaluation modes for each argument: strict-mode (S-mode) and delay-mode (D-mode). According to this paper, the algorithm is also effective for detecting an error caused by the lack of lazy evaluation (lenient cons) in stream manipulation. The algorithm is implemented in Common Lisp on a Vax.

[6] Kanekura,Hiroshi, Nishikawa,Hiroaki and Terada,Hiroaki(Osaka University): An Implementation of User-support Environment for the Ultra-High-Level Diagrammatical Specification System, Dataflow Workshop 1987, pp.41-48.

The authors emphasize the importance of graphic representation of control flow to support a specification description using topdown prototyping.

[7] Yanagi,Jun'ichirou, Nishikawa,Hiroaki and Terada,Hiroaki(Osaka University): A Study on an Implementation of the Ultra-High-Level Diagrammatical Language Processing System, Dataflow Workshop 1987, pp.49-56.

The authors emphasize the importance of a graphical specification system for dataflow computation.

Session 1B Control Mechanism

[8] Sunahara,Hideki and Tokoro,Mario (Keio University): A Note on a Half-dynamic Parallel Execution Mechanism Based on Dataflow Control, Dataflow Workshop 1987, pp.57-63.

The authors offer a good observation about how to schedule an optimal working set in a dataflow machine with hierarchical instruction memories, i.e., a primary instruction memory and a secondary instruction memory. A new analyzing technique, "shading", is introduced to analyze a scheduling policy of dataflow programs containing loops and recursion. They also propose four working set policies: an instruction segmentation policy, instruction fetch policy, replacement policy and branch management policy. The simulation results show that the policies make it possible to reduce the size of the primary memory space from one-eighth to one-eighteenth that of the MIT dynamic architecture, with only 20 percent of performance overhead.

[9] Iwata,Makoto, Nishikawa,Hiroaki and Terada,Hiroaki(Osaka University): A Study of Load and Function Distribution Schemes in a Highly-Parallel Processing System based on a "Flow-thru Processing" Concept, Dataflow Workshop 1987, pp.65-72.

The authors propose a stream-based multiple processor system. The system consists of a group of circular pipeline processors connected by a torus network.

[10] Takeoka,Shozo (ASTEC): Resource Control of a Data Flow Machine, Dataflow Workshop 1987, pp.73-77

Takeoka discusses resource control problems on a dataflow machine.

[11] Sowa,Masahiro (Nagoya Inst. of Tech.) and Hayashi,Hiroya (Gunma University): Dataflow Operating System and Indefinite Postponement, Dataflow Workshop 1987, pp.79-85.

The authors discuss problems arising in a dataflow operating system from their experience on a space-sharing system for a dataflow computer (SSS-d): interruption, input-output operations, resource management and process management on a dataflow machine. They also discuss the solution to the problem of indefinite postponement caused by nondeterministic behavior of a dataflow operating system.

(1) Interruption For interruption, the authors propose to attach a fetch precedence tag to a token so that the system can execute an interruption handler prior to an interrupted process. In their system, "normal" interruption does not prevent the execution of the interrupted process. The only restriction is that the interrupt handler must have higher priority than the interrupted process. They also propose to attach a process-ID-TAG to a token to realize such interruption that stops an interrupted process.

(2) Input-Output The authors propose two ways to realize input/output operations in a dataflow computer. One is to provide special tags that indicate input/output tokens. The other is to provide a special token-stocking area for input/output. In either way, input/output operations are performed in dataflow fashion.

(3) Resource Management For resource management, the authors discuss how to implement a historic-sensitive process, shared data and fairness to the shared data access. In the dataflow machine, the history-sensitive process can be implemented by a cyclic loop that carries a state as a token, or through the use of memory concept. In the latter case, a mutual exclusion mechanism is required to avoid non-deterministic behavior caused by the use of side-effects.

They propose two ways to realize a shared data: color hierarchy and a single-color procedure. The color hierarchy is an extended fire-rule that allows a token on an arc to match a token of the same or weaker color. The combination of the circular loop and color hierarchy makes it possible to share a data in a procedure. The combination of the memory and color hierarchy makes it possible to share a data among more than two procedures. The authors also propose a super color, the strongest color that can match any color tokens. This permits define a data shared by all procedures in a dataflow computer.

The single-color procedure is one in which every procedure call produces the same color tokens. This enables every token to match the shared data because there is only one color. Token-passing must be prohibited to ensure the determinacy of the procedure.

The authors propose four ways to realize fairness to shared data access; numbering method, time-stamp method, block-queue method and parallel-queue method. The problem results from the limitation that it is difficult to detect the order of a shared-data access in a dataflow machine. The numbering method numbers the access-tokens in order of arrival. This method is effective only when the numbering overhead is negligible. The time-stamp method deals with the oldest-stamped access-token currently available. The merit of this approach is that time-stamping can be performed on the calling procedure side rather than the shared-data side. The block-queue method groups the first

K-tokens or tokens which arrive within K constant time. Each group is served in the FIFO manner. Its merit is that it reduces the overhead of numbering. Its demerit is that "fairness" is not insured in a group. Finally, the parallel-queue method uses parallel FIFO queues.

(4) Process Management For process management, the authors discuss how to start, terminate and stop a process, and how to determine process precedence in a dataflow machine. A process in a dataflow machine has the same three states it does in conventional von Neumann computers; executable, executing and waiting. The difference is that there is a process some of whose tokens are suspended while others are executing. Such process behavior makes process management more complicated.

Starting a process is easily realized by adding initial or initializing tokens. Process termination requires special tokens that tell the operating system to terminate, because there is no way to predict the program termination in a dataflow machine. Stopping a process is very difficult to achieve, because it requires cleaning all tokens in the process. One good solution is to issue a command to all processors to stop the process. This requires the process identification mechanism described below.

Process identification can be realized by simply adding a process-identification number to tokens, or utilizing "color" which is assigned for each procedure call. The authors propose a map that manages the correspondence of color and the procedure. The problem is that this map must be accessed by all processors. That is, the information should be loaded in a common area or propagated to all processors.

The authors propose two ways to determine process precedence in a dataflow computer. The one is a precedence to change a status from executing to executable. The other has to do with the number of processors allocated to the process. The former can be realized by ordinal scheduling queues on a matching memory or a token memory. The latter can be realized by sending process-id and color information to all processors performing the process. It should be noted that assigning more processors does not always result in higher performance because a lot of processors may idle if there is not enough parallelism. To avoid an idle processor, each processor should manage several processes. In this case, however, the execution precedence will be also required.

(5) Indefinite Postponement The indefinite postponement problem occurs when there is a perpetual process like an operating system, or there are more than two jobs at any time. It does not occur when there is only one program on a dataflow machine, since the job will end sometime.

The indefinite postponement problem results from the unfair scheduling of instruction packet-making or instruction packet-execution. The indefinite postponement in the instruction packet-making phase can be protected by the following methods.

- Assuring at most one token on any arc.
- Setting a time when a processor has no waiting-token.
- Making an instruction packet in FIFO fashion.

The indefinite postponement in the instruction packet-executing phase can be protected by the following methods.

- Setting a time when a processor has no instruction packet.
- Executing an instruction packet in FIFO fashion.

Comment: This paper is a summary of the authors' previous papers published from 1985-7, unfortunately all in Japanese. I am not sure whether they have yet developed a dataflow operating system or not. Nevertheless, their work is important and worth considering, although it reminds me that a dataflow machine has to trade considerable performance for a sophisticated operating system.

Session 2 Evaluation

[12] Nogami,Shinya (NTT): Performance Evaluation of Program Execution Time in a Dataflow Control Scheme–Analysis of a Processing Element–, Dataflow Workshop 1987, pp.87-94.

Nogami analyzes the average waiting time of a matching storage and average circular time of a processing element in a dataflow machine by means of queueing network models. According to this paper, the average waiting time W is $2Q/L$ where Q is an average number of packets and L is an average arrival ratio of the packets. A dataflow model is modeled on a BCMP type queuing network, and the simulation results show the validity of the model.

[13] Sato,Masatoshi and Goto,Atsuhiro (ICOT): The Evaluation of KL1 Parallel System on Shared Memory Multi-processor, Dataflow Workshop 1987, pp.95-102.

Detailed simulation results concerning load balancing of a parallel inference machine on a Balance 21000 are reported. The results show that the main thing that prevents scalability is the lack of parallelism. The locking operation overhead, inter-processor communication overhead and suspend operation overhead do not matter in the case of the bench-mark programs, such as 8-Queens, a natural language parser (BUP), quick-sort, a prime-number generator.

Comment: The paper would be more useful if they had gotten good bench-mark programs for parallel logic programming.

[14] Kuno,Eiji, Ito,Noriyoshi and Oohara Teruhiko (Oki): An Evaluation of Stream Processing on a Parallel Inference Machine PIM-D, Dataflow Workshop 1987, pp.103-110.

The authors report the effect of stream representation of list cells in a parallel logic programming language (GHC) on a parallel inference machine (PIM-D) which contains 16 processing elements and 15 structure memories. According to this paper, a "packet-stream", rather than list structures, makes it possible to reduce the number of structure memory access operations by up to 70 percent, in the case of a prime number generator and an N-Queens problems.

Comment: The packet-stream seems to be a good way to implemente a "stream" on a dataflow machine. It is just a kind of buffer with a fill-pointer which indicates the last position of the stream. Such implementations can also apply to input/output streams.

Session 3 Architecture (1)

[15] Takai,Yoshiaki, Iha,Michiharu, Ikebe,Tadao and Shigei,Yoshiharu (Tohoku University): A Hierarchical Control Scheme of the FP Graph Reduction Machine, Dataflow Workshop 1987, pp.111-118.

The authors propose a "MISD-type" general pipeline architecture for a functional language FP. The machine consists of a reduction controller and pipeline-processors sharing a structure memory. The objective is to execute linear recursive functions in pipeline fashion. According to this paper, all FP primitive functions belong to the linear recursive function class, and can be executed in pipeline fashion.

This paper discusses the scheduling algorithm and implementation of multi-tasking on their machine. The simulation results show that the effect of pipelining cannot be

observed unless the task granularity is fairly large, because of memory access latency and the overhead incurred in finding a next task to be executed.

[16] Amamiya,Makoto (NTT): A New Parallel Graph Reduction Model and its Machine Architecture (in English), Dataflow Workshop 1987, pp.119-128.

Amamiya proposes a new parallel graph reduction model and its machine architecture on the basis of dataflow scheme. The novelty is in its use of a cell-token in a dataflow model. The cell-token represents a variable instead of data. It permits interpretation of a dataflow graph as a multi-threaded control flow graph, named datarol. The datarol overcomes several weak points in a pure dataflow computation model, such as the lack of history-sensitive computation, lazy evaluation and higher order function facilities, and the flow control overhead.

[17] Ichiko,Takao (ICOT): Integrated Design Simulation on Transmission Component-based Network Oriented Toward Non-expert Designer (In English), Dataflow Workshop 1987, pp.129-133.

Ichiko discusses a VLSI design simulation method based on the communicating sequential process model.

Session 4A Macro Dataflow

[18] Arita,Takaya and Morishita,Iwao (Tokyo University): Architecture of a Parallel Machine Oriented Toward Procedural Languages, Dataflow Workshop 1987, pp.135-142.

The authors propose a parallel very high-level language-oriented machine. The processors execute an abstract syntax tree produced by a structure editor, in parallel. The structure-editor is general enough to understand a procedure-oriented programming language classified in LALR(1). Detailed architecture and simulation results are described. The simulation results shows that the system can extract enough parallelism to keep four processors busy.

[19] Toda,Kenji, Uchibori,Yoshinobu and Yuba,Toshitsugu(ETL): Macro-Dataflow Processing on a Dataflow Machine, Dataflow Workshop 1987, pp.143-150.

The authors propose a macro dataflow model providing a "gene". According to this paper, the gene is a key idea realizing mutual communication, resource management and multiple job control on a (stand-alone) dataflow machine.

A gene is an additional tag attached to a token, which deals with precedence, interruption, suspension, etc. It works like a processor-status word in a von Nuemann computer system. A gene is created and attached to a token explicitly, and inherited from an input token to an output token, until it is removed explicitly. A token may have more than one gene, if needed.

The authors propose a gene table to realize a gene. The table contains a control field for precedence, interruption, suspension and link field for a parent color, a child color, etc. All tokens in the same function instance share the same gene in the current specification for simplicity. A gene is allocated at the function invocation and is disallocated at the end of the function. In this sense, a gene is an extension of "color". Some genes do not have link information to save the overhead of frequent gene-link updating caused by loop or recursion.

They plan to use the SIGMA-1 for the implementation of their idea, probably using micro programs. The overhead, they estimate, is 7 percent for each processor, 28 percent for each cluster for recursive factorial program execution.

Comment: It is apparent that a stand-alone dataflow machine has to support a kind of control information like "gene", although the name "gene" is somewhat misleading.

[20] Hamanaka,Naoki, Tanaka,Teruo and Muramatsu,Akira (Hitachi): Performance Evaluation on a Macro-Dataflow Machine, Dataflow Workshop 1987, pp.151-158.

The authors report the simulation results of a macro dataflow machine for numeric processing. The results show that a macro dataflow machine can achieve comparable performance with a vector machine, if a program has enough parallelism. However, the program devotes almost half its run time for inter-processor data communication.

The simulated architecture is as follows:

- The processor executes a block of instructions without branches.
- The processor consists of CMOS LSI chips with 100ns machine cycle time; four cycles per instruction.
- The system consists of 512 processor elements.

- Network latency is negligible.
- Interprocessor communication instructions cost $24+3n$ cycles, where n is the length of sending data.
- Task-switch overhead is 12 cycles.

Evaluation programs are the following:

- Differential equation program: the Jacobi method for $(1024 * 1024)$ elements.
- Incompressible Fluid Flow program: the simplified marker and cell method for $(128 * 128)$ elements.

The simulation results show that a 512 PE system achieves rates 256 and 285 times faster than a single PE system for the differential equation and incompressible fluid flow programs, respectively. In addition, the execution time of the 512 PE system is almost comparable with that of a 2G Flops vector machine. The authors also conclude that some parallel algorithms, such as the Monte Carlo, will work better on a dataflow machine than on a vector machine.

Session 4B Massive Parallel Machine

[21] Oyanagi,Shigeru, Fujita,Sumikazu, Nakamura,Sadao and Suzuoka,Takashi (Toshiba): Toward a Massively Parallel Processing System for AI, Dataflow Workshop 1987, pp.159-166.

The authors report on a design philosophy behind a parallel processing system for AI. According to this paper, they have developed a prototype system consisting of 16 PEs, and plan to make a 512 PE system using conventional microprocessors next. They also plan to make a massively parallel processing system using their own one-chip processor in future.

[22] Tsuchitani,Hajime and Takahashi,Yoshizo (Tokushima University): A Simulation of Neural Network on the Binary-tree Machine CORAL, Dataflow Workshop 1987, pp.167-174.

Evaluation of neural network simulation on a binary tree computer system (64 PEs) is reported. The authors claim that a binary tree processor system works better than a torus or hypercube network system for a neural processing. The performance itself is not so attractive because of a slow processor (MC68000) and high data-communication overhead. The paper describes a neural network model well.

[23] Yoshioka,Yoshio (Iwate University): Programmable Logic Units due to the Processing-in-memory, Dataflow Workshop 1987, pp.175-182.

Massively parallel computers based on a Programmable Logic Unit (PLU) are proposed. The PLU is a group of cells connected by a network. It works as a back-end processor or an intelligent memory for a conventional machine.

The features are summarized as follows.

- Each cell consists of registers and wired-logic processor.
- Cells are connected dynamically by mapping information from a host computer.
- No communication protocol between cells.

The basic cell consists of data registers(DR), an instruction register(IR), a result register(RR), a selector and ALU. The selector is used for selecting input data for the ALU from DRs or an input bus, and selecting an output bus to send the contents of RR.

The following variations are also proposed.

- Pipeline PLU with parallel bus lines.
- PLU array.
- PLU with time shared buses.

Comment: The novelty of PLU, though it is also a defect, is that there is no communication protocol between cells. That is, one has to wait some "cell-delay" time to get the result of PLU, and the result may change according to the computation step, and may be divergent. However, it does not have any communication overhead, and such analog-like features may be useful for some application like a neural network.

Session 5A Special Purpose Machines and Networks

[24] Akiyama,Minoru and Iriuchijima,Ken (Tokyo University): Hierarchical Software in Dataflow-computer-controlled Switching System, Dataflow Workshop 1987, pp.183-190.

A procedure level (macro) dataflow computer for a telephone switching system, named DATAFLEX-1 is reported. According to this paper, the authors have chosen dataflow architecture for their telephone switching system to make use of the transparency inherited in dataflow programming. It should be noted that they do not use a dataflow model for monitor scanning action for the reason that the action requires highly accurate real time processing.

[25] Ono,Sadayasu, Ohta,Naohisa and Fujii,Tetsurou (NTT): Dataflow Principles in Multicomputer-type DSP Systems, Dataflow Workshop 1987, pp.191-198.

The authors discuss the design philosophy of a (macro) dataflow machine, named NOV1, aimed for digital signal processing. They emphasize the importance of coarse grain parallelism for complex signal processing such as encode/decode operations of motion pictures. The NOV1 system consists of two host computers (IBM-PC), 36 PEs and Program Development Assist system (PDA) which can trace all CPU status and read/write memories via a PDA probe attached to a PE. The PDA controller also has its own host (IBM-PC), and all host computers are connected by an Ethernet and accessed from Sun-3 workstations using the NFS protocol.

[26] Kishimoto Ryozo (NTT): Development of Parallel Processing Technique in Telecommunications – A New Multi-stage Switching Network Self-routing Control, Dataflow Workshop 1987, pp.199-206.

New algorithms for self-routing multiple switching circuits are proposed. The one is a synchronous non-blocking algorithm for the Benes network. The processing time is from $\ln(n)$ to $n * \ln(n)/4$ for n switches. The other is an asynchronous blocking algorithm for the Benes network, whose latency is smaller than the fixed routing algorithm (by Huang 1984) and the random distributed algorithm (by Turner 1986).

Session 5B Structure Data

[27] Hiraki,Kei, Sekiguchi,Satoshi and Shimada,Toshio (ETL): An Efficient Structure

Handling Method for a Dataflow Supercomputer SIGMA-1, Dataflow Workshop 1987, pp.207-214.

The authors propose a new idea for the design of a structure memory for a dataflow machine. They treat a structure memory as a device that generates and absorbs a structure element sequence (they call it structure dataflow computation). Each element has its own tag to identify a data structure and an index of the element in the data structure. Structure dataflow computation are accomplished in the following steps.

- Structure dataflow creation from a High-level Structure (HLS).
- Data processing on the structure dataflow.
- Storing the structure dataflow into HLS or a Scalar High-level Structure (SHLS).

HLS represents a data structure in the structured memory. It generates a structure dataflow (sequence of elements) if it is read, attaching index information to the elements. HLS also receives a structure dataflow and stores an element according to the index of the element. SHLS is used in instances where the result is scalar rather than a structure, such as inner products.

Performance evaluation on SIGMA-1 using four PEs and four SMs shows that 1.2 MFLOPS for an inner product program, whereas 1.6 MFLOPS is the maximum performance of a PE.

Comment: It should be noted that a dataflow processor cannot get benefits unless a token carries a value, not a pointer. You can learn a lot of things from the SIGMA-1 architecture and hardware implementation.

[28] Kimura,Yasunori, Nishida,Kenji, Miyauchi,Nobuhito† and Chikayama,Takashi (ICOT,†Mitsubish
Realtime GC by Multiple Reference Bit in KL1, Dataflow Workshop 1987, pp.215-222.

The authors discuss a real time GC scheme using one bit reference counter. In this scheme, each pointer has a special bit that indicates whether the pointer is a unique pointer to the data or not. Note that the reference counter bit is attached to the pointer, not in the object. The Compiler can detect such reference information statically due to the nature of Guarded Horn Clause (GHC); variable scope is limited in a clause and structures are copied when they are read.

The simulation results show that the scheme can reclaim 100 percent of garbage cells in the case of a prime number generator program, 60 percent and 30 percent in the case of an eight-queens program and a bottom up natural language parser (BUP), respectively. The overhead for the scheme is estimated to be about 50 percent if appropriate hardware is supported.

[29] Matsumoto,Akira, Nakagawa,Takayuki, Sato,Masatoshi, Nishida,Kenji and Goto,Atsuhiro (ICOT): Parallel Cache Mechanism Optimized for KL1 Memory Access Characteristics, Dataflow Workshop 1987, pp.223-230.

The authors discuss a design for and detailed simulation results of a parallel cache for a parallel inference machine, called a five-state cache, which acts like a snoopy-cache. The difference between the five-state cache and the snoopy cache is that every processor can be a host of data-transfer in any state in the five-state cache, while a processor can be a host only if the processor has owner status in the snoopy cache.

The simulation results show that 256 columns, 4 sets, 4 words per block is optimal for a 4K word cache per processor. The other interesting characteristics of the cache are the following: the heap occupies about 85 percent of total memory capacity, but is accessed less frequently compared with other areas: only 15 percent of the total number of memory accesses. In addition, the ratio of read to write operations is almost the same, and about 40 percent of the read operations require lock operation.

Session 6 Architecture (2)

[30] Sakai,Shuichi, Ymaguchi,Yoshinori, Hiraki,Kei and Yuba,Toshitsugu (ETL) : Introduction of a Strongly-connected-arc Model in a Data-driven Single-chip Processor EMC-R, Dataflow Workshop 1987, pp.231-238.

The authors propose a system which consists of thousands of single-chip dataflow machines named EMC-R. The significant points of EMC-R are as follows. The chip does not provide associative operand matching hardware any more. It uses dynamic memory allocation for each function call, as seen in the NTT dataflow machine and the MIT dataflow machine Monsoon. The chip also provides a short-cut circular pipeline to deal with macro dataflow processing as well as instruction-level dataflow processing. The short-cut circular pipeline executes a consecutive instruction sequence exclusively, and makes it possible to implement critical sections, such as exception handlers and monitor operations, on a dataflow machine.

The authors also emphasize the merit of the short-cut circular pipeline, from the performance point of view. This is because the short-cut circular pipeline can remove the overhead of packet communication, and can reduce the total execution machine cycles per token. This is very important when a function has less parallelism. The performance estimation shows that the total processing time per packet takes 10 machine cycles for the long pipeline, and 4 machine cycles for the short pipeline. The pipeline execution times are 5 and 2 machine cycles for the long and short pipeline, respectively. This implies that the peak performance of the chip will be 100 nsec when the machine cycle is 50 micro second (20MHz).

Comment: This project is very attractive and has a lot of stimulating ideas, although the chip seems to require future devise technologies. Two papers are available for EMC-R; one is in the proceedings of the 1986 dataflow workshop and the other is in the proceedings of the 1987 computer architecture workshop, both in Japanese.

[31] Nohmi,Hitoshi, Kikuchi,Takeshi, and Nakada,Katsutoshi (NEC): The Dataflow Mechanism of NEDIPS, Dataflow Workshop 1987, pp.239-247.

NEDIPS is a back-end-type stream-oriented dataflow machine aimed for numeric processing application, developed at NEC. It should be noted that NEDIPS is the only dataflow machine commercially available in the world now. It has achieved 20-30 MFLOPS for some applications. This performance is fairly good compared to the other dataflow machines.

The significant point of NEDIPS architecture is that it supports pipeline-processors connected by a ring. Currently, the machine provides two rings; one for arithmetic operations, the other for address calculation. Each ring can have up to seven processors whose cycle time is 75nsec or 150nsec.

This paper discusses deadlock protection techniques and optimization techniques on NEDIPS. The deadlock protection is one of the important issues in dataflow programming, especially in a commercial machine. The deadlock may occur when the number of tokens exceeds the size of a FIFO buffer in a ring, or the number of unmatched-tokens exceeds the size of a matching memory. To solve these problems, NEDIPS provides automatic dataflow-control and dynamic matching-memory allocation facilities.

The automatic dataflow-control facility warns a generator or reader if a FIFO buffer (4KW) has more tokens than some specified capacity in each processor. If the buffer becomes full, it interrupts the NEDIPS control unit and saves the tokens in the buffer to a control-unit memory, then signals an error to the host machine. This simple mechanism works well and is useful for detecting a token-overflow in a FIFO buffer.

The dynamic matching-memory allocation facility enables a processor to provide a variable-length FIFO queue for each stream. The size of the queue varies from 128 words to 64 K words, and demand-driven allocation scheme is adopted to make use of the matching memory. Note that the FIFO queue is corresponding to an arc in the dataflow graph, and there is no notion of a colored-token in this architecture. Such design simplifies the token-matching mechanism and greatly contributes to the high performance of NEDIPS.

NEDIPS provides some optimization techniques to realize more efficient dataflow computation: dynamic program overwriting and a loop generator. The dynamic program overwriting is used for saving the overhead of program loading from a host machine to NEDIPS. Some special tokens change a procedure definition so that the procedure initializes only once and/or the procedure acts like another procedure with a similar algorithm, e.g., Fast Fourier Translation (FFT) and Inverse Fast Fourier Translation (IFFT).

A loop generator produces a sequence of integers when an initial value, the number of generated-tokens, and their difference are specified. This simple mechanism greatly saves the overhead of loop implementation in dataflow computation.

Comment: NEDIPS differs from the other dataflow machines in that it provides pipeline-processors and deals with only stream data. In this sense, NEDIPS is conservative and less general. However, it should be worth considering that most successful supercomputers are based on pipeline processing, that is, stream operations.

[32] Fushimi,Yoshiki, Ono,Norihiko and Onaga,Kenji (Hiroshima University): Dataflow Parallel Computing on Array of XY-double Tree Architecture, Dataflow Workshop 1987, pp.247-254.

The authors propose a tree architecture whose leaf is an IMPP (NEC's one-chip dataflow computer) and whose node is a MAGIC (interface processor for IMPPs). The tree network is doubled so that each processor can communicate more efficiently. The detailed specification of network architecture is well described.

Session 7 Software Environment

[33] Takahashi,Naohisa and Ono,Satoshi (NTT): A Declarative Debugger for Functional Programs, Dataflow Workshop 1987, pp.255-262.

The authors propose a new methodology to debug functional programs declaratively: strategic projection graph minimizing method, an elaboration of algorithmic debugging based on divide-and-query. The novelty of this work is in its use of an "applicative cache" as a filter to get debugging information as well as memorized computation. According to this paper, such information is also useful for detecting non-terminating and divergent recursive function applications. A prototype system is available on Vax lisp.

[34] Ozeki,Satomi, Kikuchi,Takeshi, Nohmi,Hitoshi and Nakada,Katsutoshi (NEC) :The Programming System and Language for NEDIPS, Dataflow Workshop 1987, pp.263-270.

The authors discuss the NEDIPS software system. The main purpose of the system is the following: natural linkage between dataflow programs and FORTRAN programs, and elimination of program-loading overhead in order to make use of the NEDIPS hardware. Dataflow programs on NEDIPS act like subroutines of FORTRAN programs on a host computer. The system provides a special macro-preprocessor and library programs for the ease of dataflow programming in FORTRAN.

The NEDIPS software system provides self-cleaning and program overwriting facilities, to eliminate program loading overhead. The self-cleaning is one that clears the side-effects of impure procedures by means of loading initial values from a host machine. This makes it possible to reuse a procedure that requires initialization before execution. The program overwriting is one that overwrites a procedure dynamically so that the procedures represents similar but a different algorithm.

The system also provides an program-overlay facility that exchange procedures dynamically to execute a large program that cannot be loaded in NEDIPS at once.

Comment: Note that NEDIPS is a commercial machine and the most users still prefer to use a familiar language Fortran, unfortunately. in addition, reducing program loading cost is as important as reducing program execution cost from the user's point of view.

[35] Okudaira,Seiji (Gunma University) and Sowa,Masahiro(Nagoya Institute of Tech.): Dataflow Language P, Dataflow Workshop 1987, pp.271-278

The authors propose a dataflow language P that can contains von Neumann-type procedures.

Session 8 Data-driven Processor

[36] Takata,Hidehiro, Komori,Shinji, Tamura,Toshiyuki, Tomisawa,Osamu, Shima,Kenji (Mitsubishi Ele. Corp.) Nishikawa,Hiroaki, Asada,Katsuhiko and Terada,Hiroaki (Osaka University): An Evaluation of Data Buffering Capability in a Data-driven Processor, Dataflow Workshop 1987, pp.279-286.

The authors discuss the simulation results of a circular pipeline architecture by GPSS. The results show that no collision occurs when a hash space dedicated for token-matching is more than 6 bits. In addition, three words are sufficient for a FIFO buffer for ALU.

[37] Yamasaki,Tetsuo, Meichi,Mitsuo, Fukuhara,Takeshi, Komori,Shinji, Shima,Kenji (Mitsubishi Ele. Corp.), Nishikawa,Hiroaki, Asada,Katsuhiko and Terada,Hiroaki (Osaka University): An Execution Mechanism for a Data-driven Processor System with the External Queue and its Evaluation, Dataflow Workshop 1987, pp.287-294.

The authors propose a revised version of a token matching management scheme based on "data-exchanging method". Their idea is to provide an external queue that maintains overflowed-tokens. In their scheme, each token has a node number and a generation number to solve hash-collision in the token-matching phase. When hash-collision occurs, the data-exchanging method exchanges the tokens so that the matching memory always holds a smaller node and generation number token. The external queue hold the other token and return it to the matching memory when the hash-collision is solved.

Comment: The authors' goal is to make a one-chip dataflow machine whose matching memory space is limited. In such a situation, their idea may be useful. However, it should be considered that it is difficult to get high performance with a complicated token-matching mechanism.

[38] Miyata,Souichi, Matsumoto,Satoshi, Azuma,Daisuke, Komatsu,Kohji (Sharp Corp.), Nishikawa,Hiroaki, Asada,Katsuhiko, Terada,Hiroaki (Osaka University) : An Implementation of Elementary Function in a Data-driven Processor, Dataflow Workshop 1987, pp.295-302.

The authors propose a dataflow processor that consists of an elastic data transmission line by self-timed circuits. They describe implementation in much detail.

[39] Okamoto,Toshiya, Yoshida,Shin-ichi, Miyata,Souichi (Sharp Corp.), Nishikawa,Hiroaki, Asada,Katsuhiko, Terada,Hiroaki (Osaka University): A Study on an Optimization of Execution Control in a Data-driven Processor, Dataflow Workshop 1987, pp.303-310.

The authors propose a dataflow architecture that supports two token-matching units.

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Resource Requirements of Dataflow Programs

Computation Structures Group Memo 280
December 15, 1987

David E. Culler
Arvind

Also appears in the Proceedings of the 15th Annual Symposium
on Computer Architecture, IEEE/ACM, Honolulu, Hawaii, May 31-June 2, 1988.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Resource Requirements of Dataflow Programs

Abstract

Parallel execution of programs requires more resources and more complex resource management than sequential execution. If concurrent tasks can be spawned dynamically, programs may require an inordinate amount of resources when the potential parallelism in the program is much greater than the amount of parallelism the machine can exploit. We describe *loop bounding*, a technique for dynamically controlling the amount of parallelism exposed in dataflow programs. The effectiveness of the technique in reducing token storage requirements is supported by experimental data in the form of *parallelism profiles* and *waiting-token profiles*. Comparisons are made throughout with more conventional approaches to parallel computing.

1 Introduction

The dataflow approach exposes ample parallelism, even in programs where little attention has been paid to special parallel algorithms, but the approach may be *too successful* in this respect. As more parallelism is exposed, more resources are required. Unless precautions are taken, programs with tremendous parallelism will saturate, and even deadlock, a machine of reasonable size. We support this claim with *parallelism profiles*, showing the number of concurrent activities over time, and *resource profiles*, showing the corresponding number of waiting tokens, on a variety of examples. This resource problem arises in any system which allows dynamic generation of concurrent tasks, although it may be most acute in dataflow models since parallelism is so aggressively exposed. Ideally, enough parallelism should be exposed to fully utilize the machine on which the program is executing, while minimizing the resource requirements of the program. We show that limiting the maximum number of concurrent iterations of loops is effective in reducing the resource requirements of typical scientific programs without sacrificing performance. The implementation of this idea is based on compiling loops into dataflow graphs with a "loop-bounding" parameter that can be set at run time according to some policy.

We begin Section 2 by articulating the resource requirements of programs on conventional, sequential machines and show that certain properties of high-level languages complicate storage management. We then argue that parallel execution generally requires more storage than sequential execution. Both of these concerns must be addressed to support a powerful, parallel programming language like Id [15, 7]. Since the execution of dataflow programs is usually given in terms of propagating data *tokens* through a graph, it is not obvious what resources are needed to execute a dataflow program, so Section 2 describes the resources associated with dataflow programs and compares these to familiar storage concepts. Section 3 develops the concept of resource profiles, expanding on the characterization of programs offered by parallelism profiles [3], and presents data for various examples. Section 4 introduces a method of controlling resource requirements by constraining the amount of exposed parallelism, and demonstrates its effectiveness. Section 5 examines broader resource management concerns, such as recycling data structures and other resources within portions of the program graph.

2 Dynamic Resource Allocation Under Parallel Execution

2.1 Stack-and-Heap Storage Model for Programming Languages

One way to characterize the expressiveness of a programming language is by its storage model. Among high-level languages Fortran has the simplest storage model. The storage requirement of a Fortran program is determined at compile-time and does not change during the course of execution. The static storage model contributes to the efficiency of Fortran at the cost of expressiveness; traditional Fortran does not support recursion. Any language that supports recursion needs a stack allocated storage model to provide storage for activation frames.

Languages such as Algol-60 and its modern derivatives support recursive procedures and, consequently, rely on a stack-allocated storage model. However, these languages do not permit data structures to be returned by a procedure, and the lifetime of an array or record may not be longer than the lifetime of the procedure activation that creates it. The reason is that such a data structure cannot be allocated on the procedure activation stack; a heap-allocated storage model is required to support dynamic creation of data structures which can be passed around freely. A storage heap is a directed graph of objects. At the implementation level, a distinction is always made between an object (a data structure) and its descriptor (a pointer). The storage occupied by a data structure is not released until it is determined implicitly or explicitly that no part of the data structure can be accessed. Pointers, on the other hand, are manipulated more like scalar values.

Allocation and deallocation of storage is significantly harder for heaps than stacks. Consequently, most language implementations treat stack and heap storage differently. Stacks are used for procedure activation frames which contain local variables and heaps are used for data structures accessible from more than one frame.

All modern programming methodologies require a heap storage model, although differences in operations on data structures permitted in various languages are considerable. For example, in Lisp, CLU and Smalltalk storage is reclaimed implicitly by a process known as garbage collection, whereas in Pascal and C heap storage is explicitly deallocated by the programmer, perhaps in recognition of the fact that automatic garbage collection is an expensive and difficult operation. Functional and other declarative languages such as Id, Miranda, ML and Prolog invariably require a heap storage model and insist on implicit storage reclamation. However, heaps in functional languages are usually acyclic graphs which can be scanned for garbage collection more efficiently than arbitrary cyclic graphs.

There are significant differences in storage requirements of programs written according to different methodologies. Generally, a more abstract programming style implies that more storage management is done by the system and less by the programmer, and that more storage is used. For example, functional languages do not permit updating of data structures and, hence, often require more storage or at least more storage allocation and deallocation operations than, say, Lisp. Among functional languages also there are significant differences between those that permit non-strict functions and data structures (*e.g.*, Miranda and Id) and those that do not (*e.g.*, pure Scheme and ML). Non-strictness supports a methodology of programming with infinite objects which often simplifies control structures in programs. In addition, it allows more parallelism to be exposed in programs. In particular, non-strict data structures allow concurrent production and consumption, and thus require synchronization on an element-by-element basis. One cost of supporting non-strict data structures is that the lifetime of data structures is often longer than in strict implementations. Estimating the "real" storage requirements of programs under automatic storage management is difficult; non-strictness makes the problem even more difficult.

2.2 Parallelism and Storage

Exploiting parallelism in a program increases its storage requirements and complicates storage management. The simplest form of parallelism is the introduction of vector operations to replace certain loops. Consider the vectorization of the following Fortran program for inner product.

```
      SUBROUTINE IP (A, B, N)
      DIMENSION A(N), B(N)
      S = 0.0
      DO 10, I= 1,N
        T = A(I) * B(I)
        S = S + T
10    CONTINUE
      END
```

Following the recipe given in the literature [14, 19] this would be vectorized by expanding the scalar T into a vector of size N and splitting the loop. The result is shown below.

```
      SUBROUTINE IP (A, B, N)
      DIMENSION A(N), B(N)
      temporary T(N)
      DOALL  5, I= 1,N
        T(I) = A(I) * B(I)
 5    CONTINUE
      S = 0.0
      DO 10, I= 1,N
        S = S + T(I)
10    CONTINUE
      END
```

Vectorization has clearly increased the storage requirement of the program. However, the subtlety here is that vector T cannot be declared locally, because its dimension is determined when the subroutine is called. It may be introduced in the same manner as A if every call is modified to pass in a temporary vector, but this requires global analysis. Temporary vectors like T could be allocated dynamically when the subroutine is called, but this fundamentally changes the storage model of Fortran. Generally, the compiler sets aside a special common block and uses it for temporary vectors throughout the program, but determining the size of this block is subtle.

With the recent availability of commercial multiprocessors, emphasis has shifted to program transformations that can spread a loop over multiple processors. One approach, developed by the NYU Ultracomputer group [12] and implemented on IBM's RP3 [16], is to maintain the loop index variable in shared memory and have each processor increment it atomically when attempting to execute an iteration of the loop. Still, each processor must maintain its own copy of local variables. In the parallel version of inner-product, each processor could have its own stack frame containing the local value of I, T and pointers to various arrays, as suggested by Figure 1. Each processor has its own local data area, in addition to the shared data area, which changes the storage model of the language in a subtle and significant manner. This approach only supports parallelization of one loop level. If a loop within a parallel loop were parallelized, an additional level of local data areas would be required, whereas the storage paradigm is one global address space and one local address space per processor.

More generally, if subroutines can be spawned dynamically as tasks, which may in turn spawn tasks, a tree of stack frames is required, not just a stack, so allocation and deallocation are no

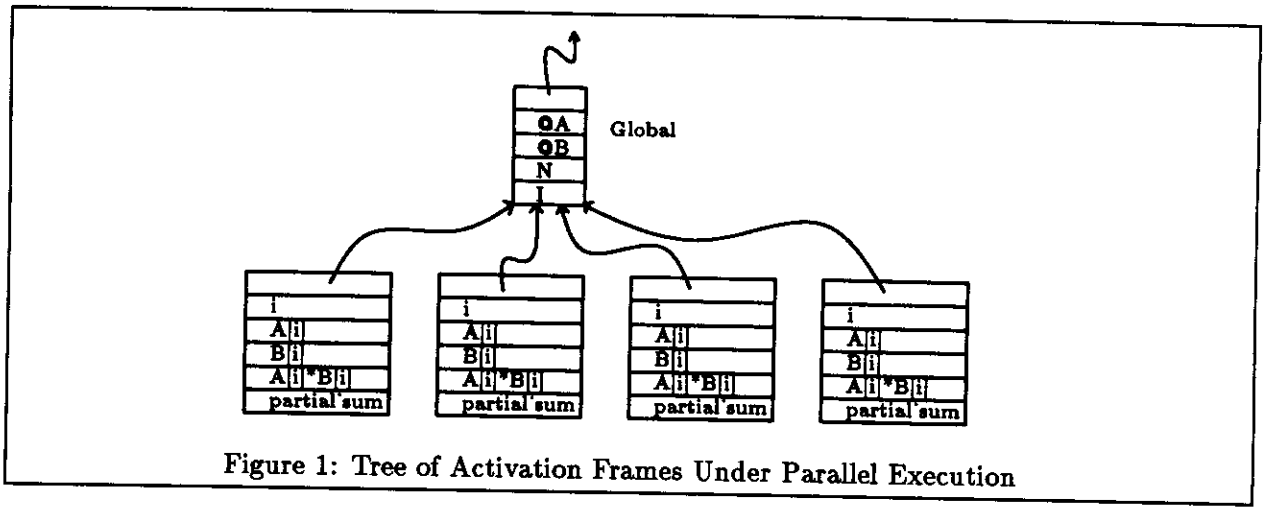


Figure 1: Tree of Activation Frames Under Parallel Execution

longer simple, constant time operations. Management of trees is only slightly simpler than general heap management, and it has been proposed that multiple tasks can be supported more uniformly by allocating local variables on the heap as well [13]. The tricky part is, of course, reclaiming unused storage in the heap, and especially doing so in parallel. Moreover, parallel execution may require exponentially more storage than sequential execution, because where the calling depth is n, the size of the tree is 2^n if each task spawns two subtasks.

Parallel execution and storage management are related in an even more subtle way involving names for synchronization points. The easiest way to explain this is by example. Take a Load/Store architecture like the Cray which is able to issue a memory read and continue to execute instructions while the read takes place. To do this, a register is used as the target of the read, and instructions can be issued as long as they do not refer to this register. Clearly, the number of concurrent read requests is bounded by the number of such registers. The register serves as a synchronization point for two asynchronous events. A similar situation arises with semaphores, locks and other high-level synchronization mechanisms — as more parallelism is exploited, more concurrently active synchronization points are required.

2.3 Storage in the Dataflow Model

The resources involved in executing an Id dataflow program are influenced by all the factors discussed above, since Id is a powerful functional language with non-strict arrays and supporting highly parallel execution. Not surprisingly, this requires a tree of activation frames and a heap. Each user-defined function and loop in the program is represented by a dataflow graph called a *code-block*. At any point, an executing dataflow program comprises a tree of concurrent code-block invocations, each identified by a *context*. Loop code-blocks may unfold allowing many iterations to be active concurrently, so the invocation tree can branch with arbitrarily large degree.

Variables in the dataflow program do not denote storage directly, but rather denote arcs in the graph. While a token is present on the corresponding arc, however, a certain amount of storage is implicitly required to represent the data value. Thus, one measure of the storage requirement of a dataflow program is the number of tokens in existence. We can consider the tokens associated with a context to be the activation frame for the code-block invocation. However, unlike stack frames which are laid out statically by the compiler, token storage is typically managed dynamically by the hardware, *c.f.*, the description of the waiting-matching stores in [1]. Often we distinguish tokens which are waiting for a partner from those that are not, because the first occupy more crucial

storage resources. Waiting tokens correspond to temporary variables held on the stack, whereas short lived tokens are more akin to values held in high-speed registers.

I-structures are write-once arrays, which are allocated dynamically, filled by the program, and implicitly reclaimed. They can be passed into and out of functions, so their lifetime is independent of the code-block invocation that creates them. Thus, I-structure storage corresponds with the heap in many language implementations. Element-by-element synchronization is provided to allow concurrent production and consumption of data structures.

In the remainder of the paper we focus on the token storage requirements of programs, as this is a critical resource in dataflow machines. Once the token store is exhausted, a dataflow machine will deadlock. I-structure requirements are, of course, extremely important as well, for generally large scientific problems are limited by storage for arrays, not temporaries. We will show, however, in Section 5, that the techniques for reducing token storage are an essential step toward solving the harder problem of effectively managing I-structures.

3 Resource Profiles of Dataflow Programs

Since a dataflow program specifies only a partial order on the set of operations it comprises, there are many different permissible schedules of operations. Different schedules may exhibit different amounts of parallelism and different resource requirements, although all schedules compute the same result. It is this flexibility that makes the approach tolerant of unpredictable memory latency [6]. The maximum token requirement of a dataflow graph can be formulated as a maximal cut problem [11], however, we have found it more insightful to estimate typical storage requirements based on a particular, well-defined execution order corresponding to an "ideal" dataflow machine. Our execution model assumes unbounded processors ($p = \infty$), unbounded storage resources, unit time operation, and zero communication latency ($l = 0$). In each step, all enabled operations execute concurrently producing results for the next step. This model and a variety of extensions to it are discussed in detail elsewhere [3]. Relative to this model, we define the *Parallelism Profile* of a program to be the number of concurrent operations over time. Similarly, the *Token Storage Profile* of a program is the number of tokens in existence during each step of the ideal execution. In later sections, we consider the behavior of programs on a more realistic model with finite processors and non-zero communication latency.

As a simple example, consider a program to compute the inner-product of two vectors. Figure 2 shows the dataflow graph in a slightly simplified form (top), its parallelism profile (middle) and token storage profile (bottom). The parallelism profile is annotated with the names of the nodes fired in each step, and the token storage profile with arc names for each token, represented by the destination node number concatenated with l or r as needed. Initially three tokens are input, one carrying the initial value of **sum** and two carrying the initial value of `i`. The $\leq$ predicate is enabled and fires, producing two tokens, one for each switch node, to make a total of four. The two switches fire, the left producing a single token while the right produces three. At this point nodes 4, 5, and 6 are enabled and a token is waiting at the left input to the +. The rest of the token storage profile is obtained by employing the dataflow firing rule and observing that when a node fires the total number of tokens changes by the number of outputs minus the number of inputs. The maximum of the token profile is the token storage requirement of the program.

Under this ideal execution model the average parallelism approaches 8/3 operations per step, while the token storage requirement is five tokens. Thus, approximately two tokens worth of storage are required to support each operation. It is not surprising that the token storage requirement is proportional to the exposed parallelism, as indicated here, since a certain amount of storage is

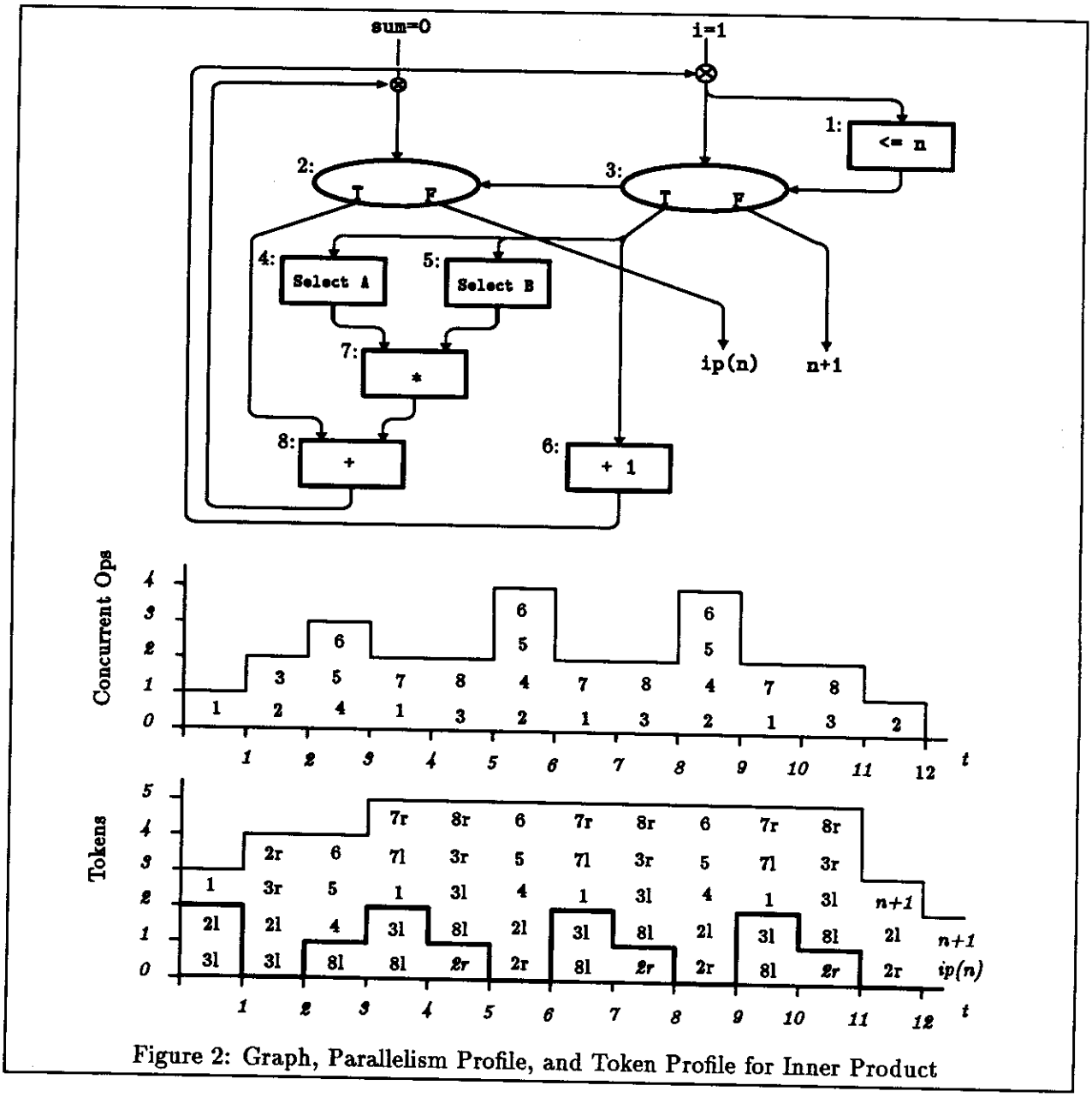


Figure 2: Graph, Parallelism Profile, and Token Profile for Inner Product

required to hold the operands of each of the concurrent activities, although we shall see later that the storage requirements can in fact be worse.

The *waiting token profile* is related to the token profile. Observe that the token produced by the left switch must wait until the result of the multiply is available. Similarly, the data input to the right switch must wait until the predicate is evaluated. The boolean input to the left switch must wait for the data input. Stepping through the idealized execution yields the waiting token profile shown by the thick line in the bottom portion of Figure 2. As these waiting tokens are architecturally most significant, we focus on waiting token profiles.

The analysis above assumes that arrays A and B are available. Suppose instead, that only array A is available, and the elements of B have yet to be filled in. Select operations against B are deferred until the stores are performed, but the index variable i continues to circulate causing elements of A to be selected, and resulting in as many as n tokens waiting at the left input to the * node. Tags associated with the tokens ensure that correct pairing is achieved with elements of B, and when the elements of B are available all the multiplies may potentially execute in parallel [5]. This scenario is quite similar to the vectorization example discussed above, and pays a similar storage penalty. Since the summation is still serial, the storage requirement is $\mathcal{O}(n)$, while the average parallelism is constant.

We now present three examples which will be used in the remainder of the paper. The first is matrix multiply using the straight-forward algorithm for which the Id code is given below.

```
DEF matmult A B r m n =
  { C = matrix ((1,r),(1,n));
    {FOR i FROM 1 TO r DO
       {FOR j FROM 1 TO n DO
         s = 0;
         C[i,j] = {FOR k FROM 1 TO m DO
                     NEXT s = s + A[i,k] * B[k,j];
                   FINALLY s}}};
  IN C};
```

It generates the parallelism and waiting-token profiles shown in Figure 3 with $r = m = n = 16$. The outer two loops unfold, allowing n^2 inner products to proceed in parallel for matrices of dimension n, and the waiting-token requirement is proportional to the amount of parallelism. The parallelism profile shows a bell shape because each loop spawns instances of inner loops in a staggered fashion. The second is a hydrodynamics and heat conduction application (Simple) run for eight time-steps on a 16x16 mesh (see [4] for details), which generates the nearly periodic parallelism profile in Figure 4 with a stair-step waiting-token profile. In this case, the storage requirements are excessive compared to the exposed parallelism. This example shows eight iterations on a 16x16 mesh; real problems involve 100,000 iterations on a 100x100 mesh. Finally, Figure 5 shows the parallelism and waiting token profiles of a 2-dimensional Gaussian relaxation for four iterations on a 16x16 grid; the parallelism and waiting token profiles are both periodic. Again, the waiting-token requirement is proportional to the amount of exposed parallelism.

4 Controlling Resource Requirements

4.1 Excessive Parallelism

Suppose we execute the 16x16 matrix multiply example discussed above on a 50 "processor" machine. It is clear from the parallelism profile in Figure 3 that the program has more than sufficient

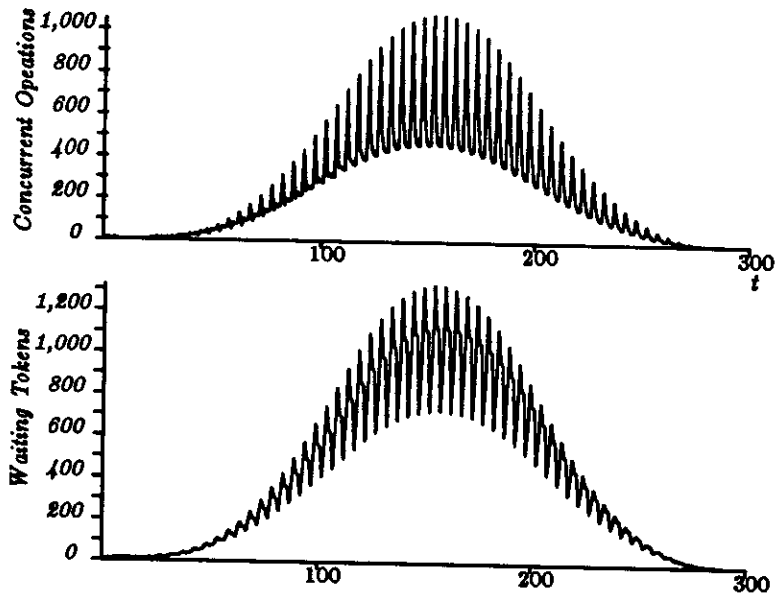


Figure 3: $(p = \infty, l = 0)$ Parallelism Profile and Token Profile for 16x16 Matrix Multiply

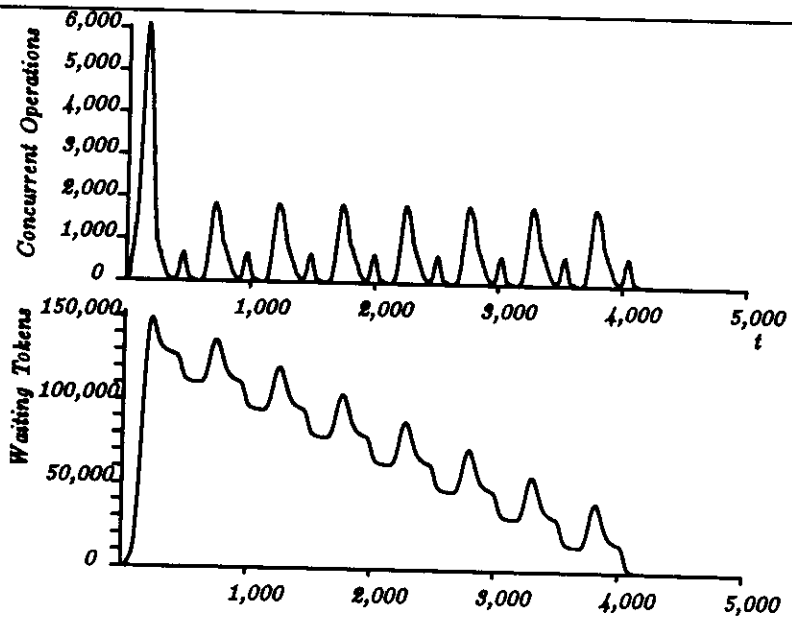


Figure 4: $(p = \infty, l = 0)$ Parallelism Profile and Token Profile for Simple, 8 iterations on 16x16 mesh

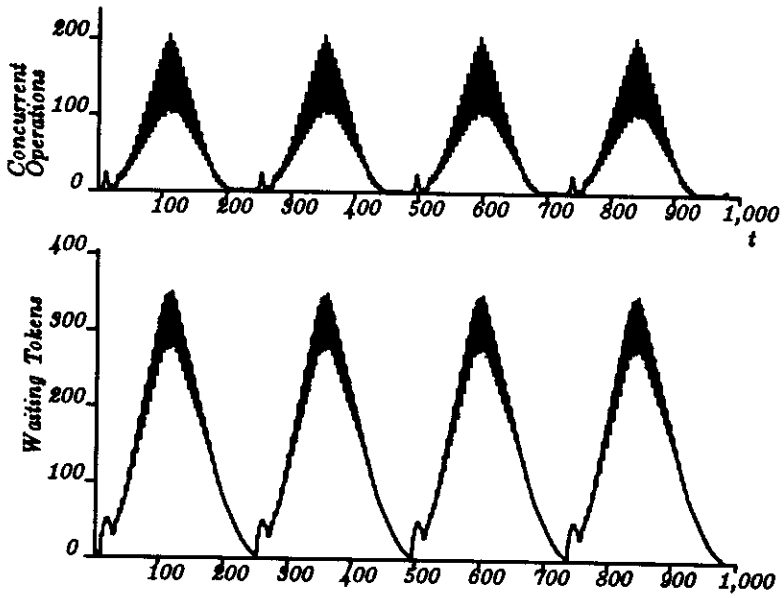


Figure 5: ($p = \infty, l = 0$) Parallelism Profile and Token Profile for Gaussian Relaxation, 4 iterations on a 16x16 grid

parallelism for 50 processors under our ideal, zero latency execution model. An important question is how does the token storage required to execute the program change when the number of operations that can be performed in a step is limited. It would be comforting if reducing the number of processors reduced the resource requirement, but Figure 6 suggests that this is not the case. With at most 50 operations per step and operations scheduled from the set of enabled operations on a FIFO basis, the number of steps required increases to 1,511, and the waiting token storage requirement is undiminished. In fact, it has increased substantially! This behavior is not particular to the matrix multiply example, we observe it universally.

Limiting the number of operations performed in each step essentially constrains the set of execution schedules, but among this remaining set are schedules with resource requirements that are close to that of the ideal model, schedules that are much better, and others that are much worse. Under fair scheduling with a finite number of operations per step the program behavior, including the resource requirement, will always be similar to that under the ideal model because processor resources are effectively multiplexed across the set of enabled activities. A variety of "unfair" schedules can be implemented, such as LIFO scheduling among enabled activities, and will result in reduced resource requirements for some programs and substantially increased requirements for others. On complex programs, the effects of such heuristics are unpredictable. It appears that there is a significant cost associated with exposed, but unexploited, parallelism.

To reliably reduce the resource requirements of a program we must constrain its execution so that less parallelism is exposed. The source of parallelism in this example is the unfolding of loops; the outer loop initiates n instances of the middle loop, each of which initiate n inner products, for $n \times n$ matrices. By limiting the number of concurrent iterations of various loops we limit the exposed parallelism. If iterations of the outer loop were executed serially, the $\Theta(n^2)$ parallelism should be reduced to Θ(n) parallelism, with the critical path extended from Θ(n) to $\Theta(n^2)$. Serializing the middle loop should have a similar effect. The inner loop is nearly serial already because of how the inner product is coded. It is conceivable to constrain any of the loops to any fixed number

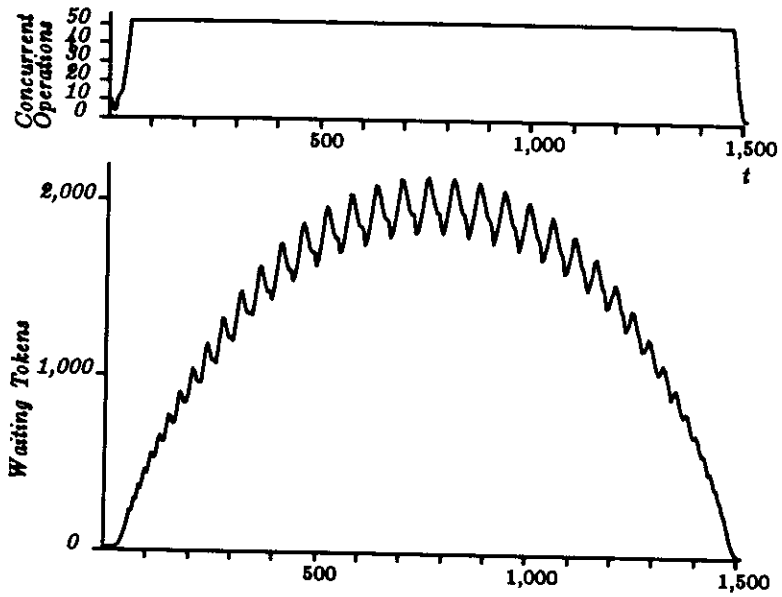


Figure 6: ($p = 50, l = 0$) Finite Processor Parallelism and Token Profiles for 16x16 Matrix Multiply

of concurrent iterations. Figure 7 shows the parallelism and waiting token profiles for the 16x16 matrix multiply example when the middle loop is constrained to two concurrent iterations and at most 50 operations are performed in each step. The resource requirement is reduced by a factor of five relative to the unbounded case, while the total number of steps increases less than 1%, (see Figure 6). Section 4.2, below, shows how this is accomplished.

This problem of excess parallelism reflects a tension between providing the programmer with a powerful, parallel programming model, where parallelism in programs is expressed without reference to the machine configuration, and effectively using machine resources. It arises with any capability of dynamically spawning tasks, and to a lesser degree with more primitive capabilities for parallel execution, such as a fixed number of tasks operating on a work queue. In order to exploit the full parallelism in matrix multiply using Fortran annotated for multiple processors, an inner product would have to be initiated on each of n^2 processors; this requires $o(n^2)$ storage in stack frames, which is comparable to what we see in the waiting token profile. Most implementations of this sort of DOALL parallelism allow each physical processor to perform at most one iteration at a time, which effectively bounds the number of concurrent iterations. In general, it is difficult to exploit parallelism in nested loops with these approaches, although in this case it is possible with a certain amount of additional control overhead. If only a small number of processors were available, this would be wasteful, so we might elect to parallelize only one loop, forcing the others to execute sequentially. Unfortunately, having done so, the program is no longer well-suited to a large number of processors. In addition, the loop we have chosen to parallelize may have few iterations, depending on the shape of matrices passed to the matrix multiply routine, resulting in little parallelism. Vectorization is difficult to apply in this example, because it must work from inner loop outward, but we observe that in order to reduce the amount of temporary storage introduced it is possible to "strip-mine" the loops, *i.e.*, break a loop into a loop of smaller vectorized loops.

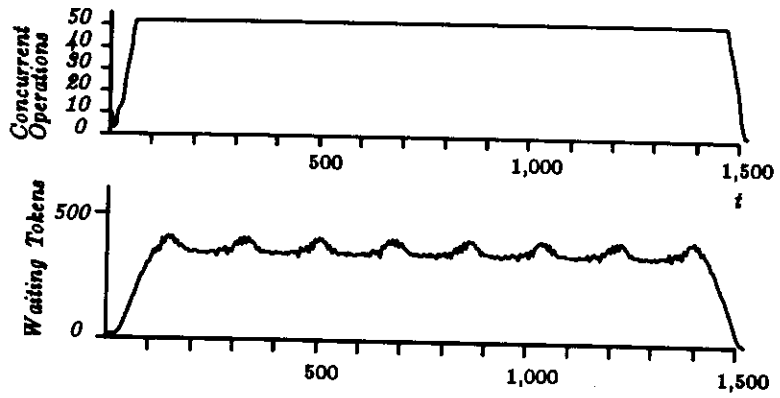


Figure 7: $(p = 50, l = 0)$ Profiles for 16x16 Matrix Multiply with Middle Loop Bound to two

4.2 Loop Bounding in Dataflow Programs

Under a dataflow approach, parallelism can be constrained by bounding the unfolding of loops in a flexible manner, which does not "compile in" a limit on the parallelism that can be exploited. The bounds can be determined at run-time, reflecting the problem size and number of processors. In a dataflow model, the only way to ensure that one event follows another is to introduce an (artificial) data dependence between the two events. It has been shown that loops have bounded unfolding under any execution order if and only if all loop variables are mutually dependent [11]. We can use this result to develop a loop schema with parameterized unfolding, *i.e.*, no more than some k iterations can be active concurrently and k may be determined at the time the loop is invoked. One such bounded loop schema, presented in [2], is shown in Figure 8. A gate is placed on the output of the predicate, which inhibits new iterations from starting unless a trigger token is present at the control input to the gate. Initially, k such trigger tokens are provided, and one is consumed each time an iteration starts. A termination tree detects when an iteration has terminated and supplies the trigger token which enables another iteration to start. In effect, initiation of a new iteration is made dependent on the termination of the one k earlier. Exact details of how tags for these tokens are manipulated and how the graphs are made self-cleaning are beyond the scope of this paper [8, 10], but we may consider the effect of this mechanism.[1]

To see the behavior of programs under various strategies for bounding the loops we return to the ideal execution model. Figure 9 shows the parallelism profiles for matrix multiply example under the ideal $(p = \infty, l = 0)$ model, with only the outer loop constrained (top), with only the middle loop constrained (middle), and finally with both loops constrained (bottom). Even though no constraints are placed on the number of operations per step, these transformed programs exhibit moderate, uniform potential parallelism. The waiting token requirements are similarly reduced, relative to the unbounded version, as indicated by Figure 10. The reduced waiting token requirements and moderate potential parallelism of these transformed programs, make them well-suited for a machine of, say, 16 or 32 processors. The best setting of loop parameters for a given invocation of matrix multiply depends on the shape of the matrices and the availability of resources. Figure 7, above, shows the behavior of the version with the middle loop bounded to two with at most 50 operations per step. Comparing this with unbounded version in Figure 6 we see how important it is for the

[1] Bounding loops introduces a degree of strictness in the computational model, and as such changes the semantics slightly. It is possible to write programs which will terminate only if arbitrarily many iterations can be active concurrently, but these programs are rather peculiar and could not be written in any conventional language.

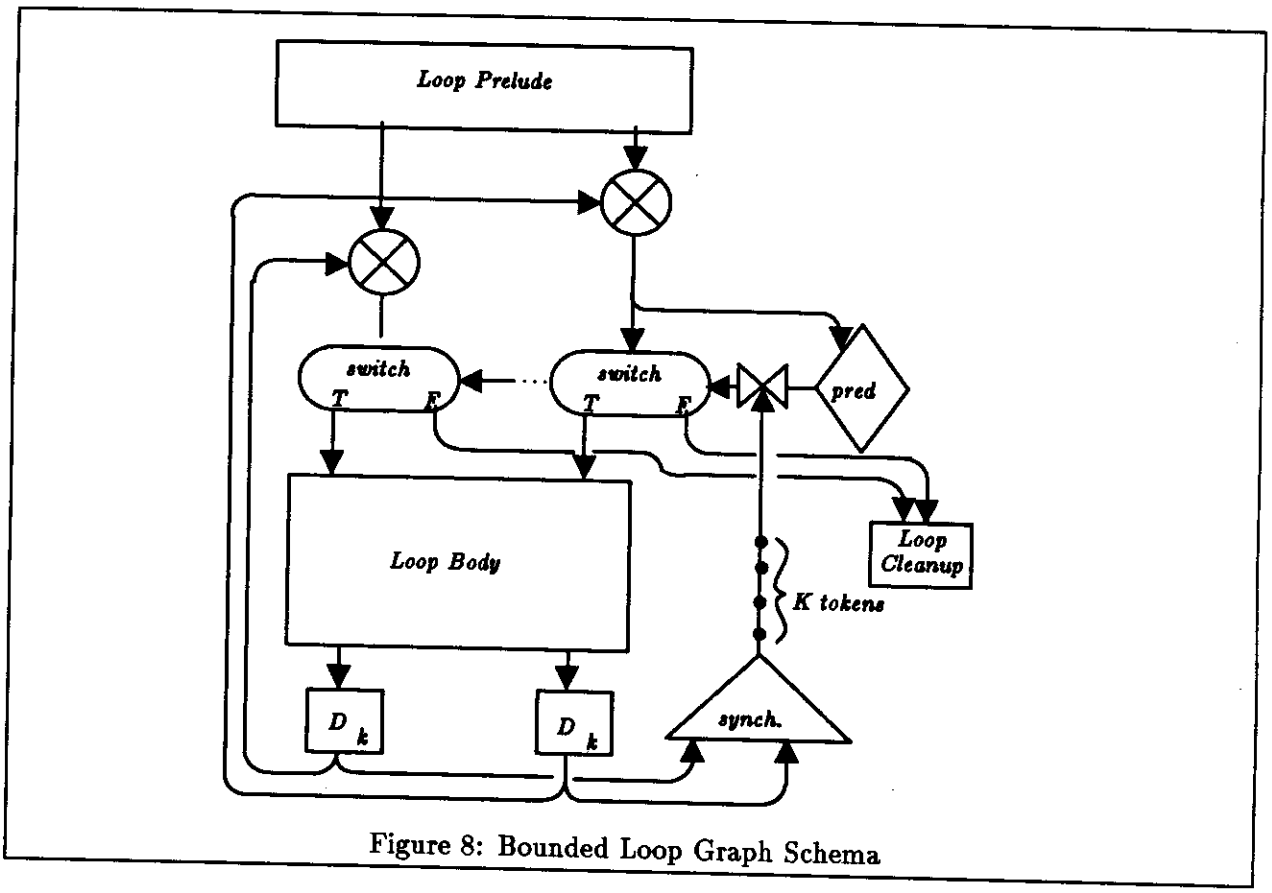


Figure 8: Bounded Loop Graph Schema

exposed parallelism in the program to match what the machine can exploit. As the size of the problem increases, the effect becomes more dramatic. By bounding all the loops, the token storage requirement of matrix multiply becomes independent of the problem size.

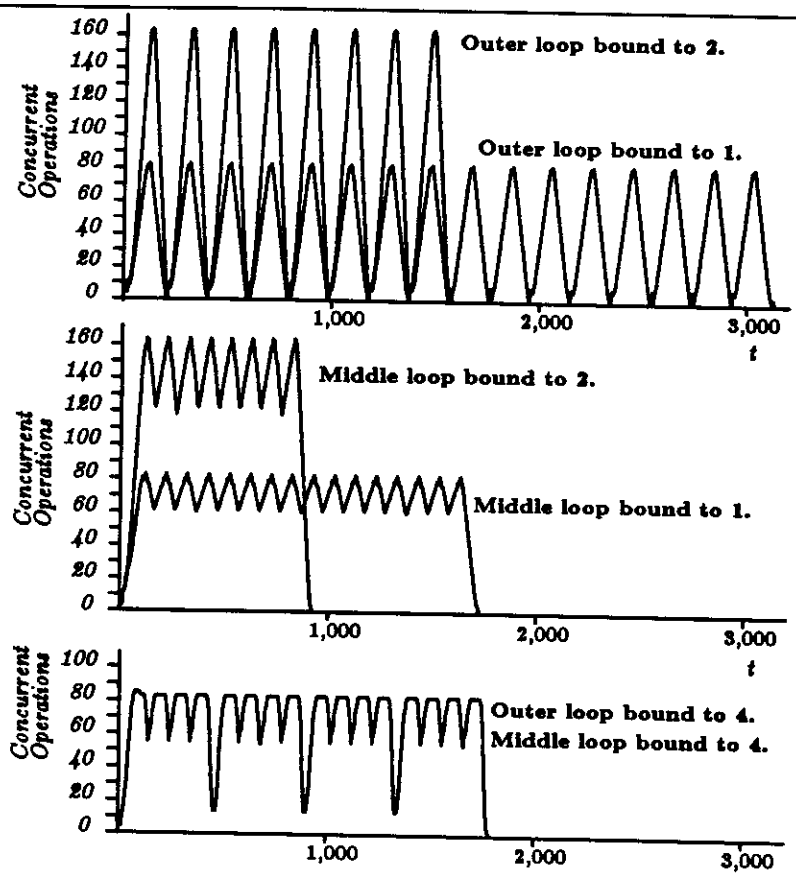


Figure 9: ($p = \infty, l = 0$) Parallelism Profiles for 16x16 Matrix Multiply Under Various Loop Bounds

The current version of the Id compiler generates parameterized bounded loops for all loop code blocks [18]. If the bound is large enough, no restriction is placed on program unfolding. This is how all the unbounded profiles in this paper were generated. Profiles with bounded loops are generated by setting loop bounds for certain code blocks prior to execution.

Under a model where iterations are distributed over processors and each processor executes one iteration at a time, the number of concurrent iterations is effectively bounded by the number of processors. It might be possible to effect a similar mechanism in a dataflow machine, but here we have expressed control of parallelism without stepping outside the formal model. Any execution model which treated each inner product as a task and scheduled them fairly would experience similar resource requirements to what we see in Figure 3.

4.3 Useless Parallelism

The profiles for Simple in Figure 4 above raise a more serious concern than the excess parallelism in the previous example — *useless parallelism*. The program unfolds, but major portions of it allocate resources and then wait until data becomes available. The eight iterations of the outer loop are

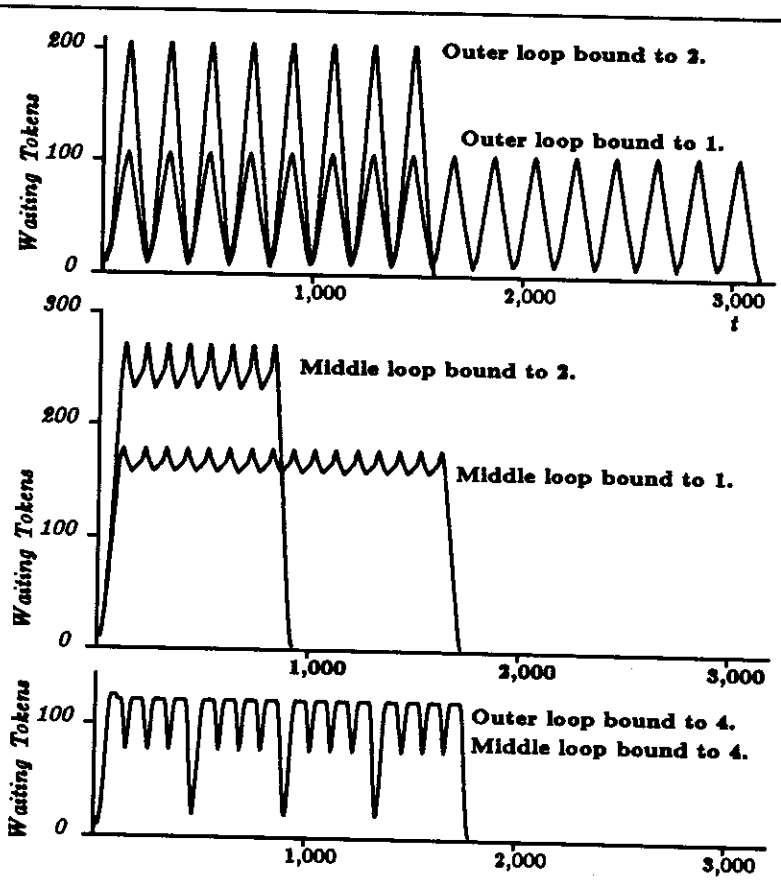


Figure 10: ($p = \infty, l = 0$) Waiting Token Profiles for 16x16 Matrix Multiply Under Various Loop Bounds

clearly visible as there is a strong constriction point, where the new time-step is determined, between iterations. Each iteration has two distinct phases, the first is the hydrodynamics calculation, the second heat conduction work. Each phase reduces the values in the mesh to a single value used in determining the time step for the next phase. Thus, each point of the mesh in one iteration depends on the entire mesh produced in the previous iteration — there is almost no useful overlap between iterations. The outer loop simply performs a number of time-steps. All eight iterations unfold, each allocates instances of the mesh, unfolds over the dimensions of the mesh and issue reads against the mesh of the preceding iteration. As the data is filled in, the deferred reads are satisfied, and the computation sweeps across the successive iterations. In many applications this kind of unfolding would expose essential parallelism, but here it serves only to increase the resource requirements.

Bounding the outer loop of this application eliminates this problem, giving the ideal parallelism and resource profiles shown in Figure 11. The waiting token requirement is now independent of the total number of iterations and proportional to the amount of exposed parallelism. The critical path is only increased by 10% even under the ideal model. For finite processors the difference is even smaller, and loops can be further constrained to reduce excess parallelism. Figure 12 shows the waiting token profiles with the outer loop unbounded and bounded to 1 under finite processor execution with 32 operations per step and communication latency of 10 units.

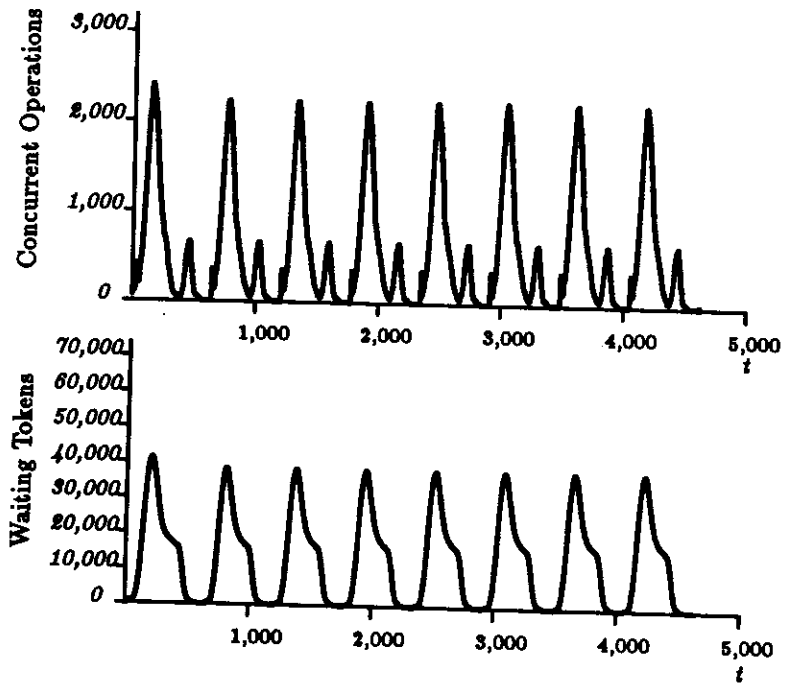


Figure 11: ($p = \infty, l = 0$) Profiles of Simple, 8 iterations on 16x16 mesh, with the Outer Loop bound to one

4.4 Inherently Bounded Loops

In many cases the dependencies within the application create inherently bounded loops. The Gaussian Relaxation example appearing below illustrates this situation as indicated by the profiles in Figure 5. The relaxation is performed until a given convergence criterion is met. We have used the maximum change of any point in the grid relative to the previous iteration; thus, the predicate of

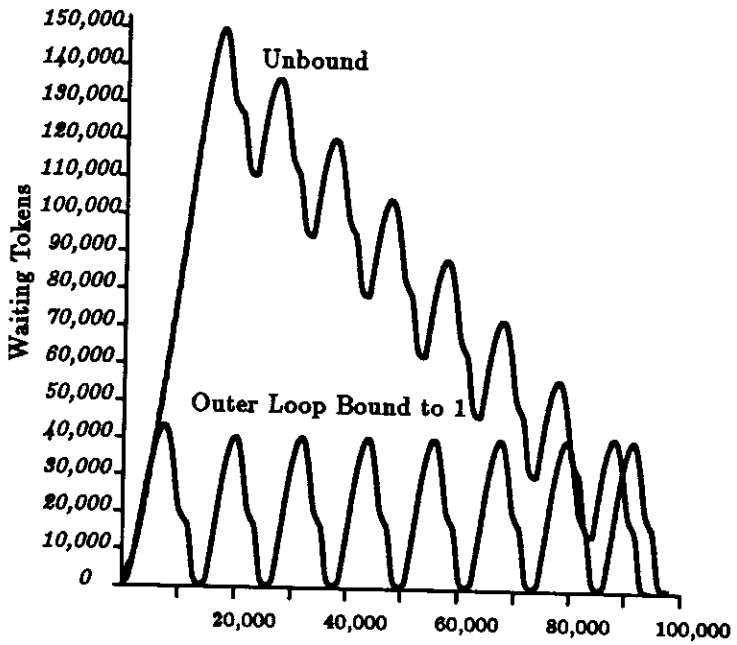


Figure 12: ($p = 32, l = 10$) Token Profiles of Simple, 8 iterations on 16x16 mesh, with the Outer Loop unbound and bound to one

the loops is dependent on every point in the grid. There may be a slight overlap between iterations because of termination detection, but the useful work of successive iterations does not overlap. It is not surprising that the waiting token profile is periodic and the storage requirement is independent of the number of iterations. Detecting this situation is important, because it ensures that a small unfolding parameter will not compromise the performance of such a program, thereby eliminating one variable in the resource management policy.

```
DEF 2D_relax A steps epsilon =
  {delta = 2*epsilon
    IN {WHILE delta > epsilon DO
      NEXT A = make_matrix (bounds A) (nine_point_stencil A);
      NEXT delta = max_pointwise_difference A (next A);
    FINALLY A}};
```

This example also shows how subtle differences in program structure may have dramatic effects on program behavior. If the result of the maximization were used in computing the new grid but did not appear in the predicate, the program would be ill-behaved in the same way as Simple. If this result were not used in computing the new grid, the behavior would be more like matrix multiply; all three loops would unfold, generating a wave-front of parallel activity. However, if we wished to extract more parallelism than indicated by Figure 5, while still testing for convergence, it would be necessary change the program, allowing a number of iterations between convergence tests.

5 Broader Resource Management Concerns

The loop bounding technique has been introduced as a means of controlling program unfolding to reduce token storage requirements, but properly generalized it may serve a broader role. In

embellishing the graph to control unfolding, we begin to build resource management into the graph itself. This can be extremely effective, because management of resources is integrated with the very structure of the program. Each iteration of a loop requires certain resources, and the trigger token fed into the gate is effectively a ticket for those resources. A block of resources can be granted to the entire loop invocation and managed locally by simply recycling resources in the graph.

5.1 Recycling Data Structure Storage

Loop bounding plays an essential role in managing I-structure storage, as well as token storage. Suppose that implicit storage reclamation were implemented by whatever means: mark-and-sweep, reference-counting, volatility regions, or any other. An I-structure can be reclaimed only when no references to the structure exist. Consider the Simple program discussed above, which is structured like many scientific applications. In each iteration, a large mesh is allocated and filled using the mesh of the previous iteration. For large-scale scientific applications we expect a few versions of the mesh to fill the entire data structure storage. Without loop bounding, the iterations unfold so that all the versions of the mesh are allocated, and references exist for each. Only as data moves through the sequence of meshes will any structures be reclaimed. Thus, the I-structure storage profile will have the same shape as the waiting-token profile in Figure 4. No reasonable machine could support 100,000 versions of the mesh at one time! With loop bounding, old versions of the mesh are reclaimed as new ones are allocated.

This idea can be carried further to recycle I-structures within the program graph. Traditional methods update the mesh in place, but this can be tricky, especially under parallel execution. The Id style is to allocate a new mesh during each iteration and to fill it in. We argued above that loop bounding is essential in this situation, because it controls the number of concurrent versions of the structure. In addition, with bounded loops the lifetime of such structures is clearly defined, so they could be reset and reused. A fixed number of versions of the mesh can be allocated initially, and recycled within the graph. In effect, this provides the kind of multiple buffering used for concurrent I/O within parts of the program, but without major programming effort.

5.2 Tag Management

Another important resource, specific to dataflow implementations, is tag space. The tag carried on a token includes an iteration identifier so that tokens belonging to different iterations do not interact. In abstract formulations of dataflow such as the U-interpreter [5], management of tags is completely localized, but the tag space is arbitrarily large and very sparsely used. Any realistic implementation of this model must restrict that tag to some fixed size. With bounded loops, iteration identifiers are represented modulo the bounding parameter, so by picking a maximum loop unfolding tags can be of fixed size. By building resource management into the graph, localized management is regained, while using the tag space densely.

Loop bounding could play an important role in existing dataflow machines such as the NEC IPP. That machine supports a limited FIFO dataflow model [1], where at most 16 tokens can queue on an arc. Graphs are carefully hand-tuned so that this limit is not exceeded. Bounded loops provide a basis for systematic code generation, making implementations of high-level languages possible on such a machine.

The notion that each iteration has a certain amount of resources with which to work suggests a novel approach to implementing token storage. Rather than perform an associative match or hashing to determine if a partner token is present, assign a block of token storage to each iteration. Tokens can be assigned to slots in the block at compile-time by coloring the graph so that no

two tokens which could potentially coexist fall in the same slot. A lower bound on the number of slots can be found via linear programming, and actually performing the assignment is similar to sophisticated register allocation schemes. [9] Setting up a loop requires allocating as many such blocks as the loop bound permits. Determining if a partner is present only requires checking a presence bit.

5.3 Controlling Unfolding in General

This loop bounding approach to controlling parallelism is applicable to a restricted class of programs. For general recursive programs there is need of similar techniques. The general approach of introducing auxiliary arcs to enforce artificial dependencies still applies, but it is less efficient to implement and less clear how to apply. Researchers at Manchester University have suggested that rather than introduce artificial dependencies in the graph, the effect can be achieved by deferring context allocations when the machine appears busy [17]. Unfortunately, deferring a context allocation does not inhibit the requester from making additional requests. Simple provides a good example. Each iteration invokes numerous code-blocks to perform various parts of the two phases of the computation. Assuming all of these invocations were deferred, the outer loop would completely unfold, making a huge number of requests. Eventually, some deferred request will be serviced and a subordinate code-block invoked, but unfortunately almost all the deferred requests would be useless to service. Only those from the first iteration, and only the hydrodynamics portion of the first iteration will result in tangible progress. Deferring requests can actually shift the resource requirements from that suggested by fair scheduling to worst-case. Certainly, if code-blocks and data structures are strict this kind of approach would be much more effective, but much of the potential parallelism in programs would be lost. Combining such heuristics with loop bounding may bring us a long way toward automatic control of parallelism of dataflow programs in general.

6 Conclusion

We have shown that parallel execution generally requires more resources than sequential execution, and complicates resource management. Dynamic generation of parallel tasks can easily "run away", generating inordinate resource demands. Some mechanism for controlling parallelism is necessary, so that enough parallelism is exposed to saturate the machine, while minimizing resource demands. We have presented one mechanism in the context of dataflow models, which allows resources to be recycled within the program graph. Given such a mechanism, the open question is what policy would be effective over a broad class of programs. Compiler analysis is part of the answer, as dependence analysis is essential to detecting useless parallelism in cases like Simple, and run-time heuristics based on availability of resources hold potential. A planning strategy based on building *resource expressions* for portions of programs has been proposed [2]. All three approaches are still subjects of study. We are confident that the techniques developed here will allow large-scale scientific programs to be executed effectively on dataflow machines.

Acknowledgments

The authors would like to thank Kattamuri Ekanadham, Gino Maa, and Steven Brobst for their contributions to early drafts of this work, Natalie Tarbet and Ken Traub for the polish they added, and all the members of the Computation Structures Group for the intellectual climate they create.

References

[1] Arvind and David E. Culler. *Dataflow Architectures*, pages 225–253. Volume 1, Annual Reviews Inc., Palo Alto, CA, 1986. Reprinted in Dataflow and Reduction Architectures, S. S. Thakkar, IEEE Computer Society Press, 1987.

[2] Arvind and David E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester, England*, North-Holland Publishing Company, July 15-18 1985.

[3] Arvind, David E. Culler, and Gino K. Maa. *Parallelism in Dataflow Programs.* Technical Report Computation Structures Group Memo 279, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987.

[4] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987. To appear in the Journal of Parallel and Distributed Computing.

[5] Arvind and K. P. Gostelow. The U-Interpreter. *COMPUTER*, 15(2), Feburary 1982.

[6] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25-29 1987.

[7] Arvind, R. S. Nikhil, and K. K. Pingali. *Id Nouveau Reference Manual, Part II: Semantics.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[8] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259)*, Springer-Verlag, June 15-19 1987. To appear in IEEE Transactions on Computers.

[9] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[10] David E. Culler. *Effective Dataflow Execution of Scientific Programs.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, August 1988 (expected).

[11] David E. Culler. *Resource Management for the Tagged-Token Dataflow Architecture.* Technical Report TR-332, Massachusetts Institute for Technology, Laboratory for Computer Science, January 1985.

[12] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Coordinating Large Numbers of Processors. *ACM Transactions on Programming Languages and Systems*, 4(1), January 1982.

[13] Robert H. Jr. Halstead. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the ACM Symposium on Lisp and Functional Languages*, August 1984.

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceeding of ACM Symposium on Principles of Programming Languages*, January 1981.

[15] Rishiyur Sivaswami Nikhil. *Id Nouveau Reference Manual, Part I: Syntax.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[16] G. F. Pfister and et al. The IBM Research Parallel Processor Prototype (RP3). In *Proceedings of the 1985 ICPP*, August 1985.

[17] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine.* PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[18] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture.* Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

[19] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.