ASYNCHRONY IN PARALLEL COMPUTING:
FROM DATAFLOW TO MULTITHREADING*

JURIJ ŠILC[†], BORUT ROBIČ[‡], AND THEO UNGERER[§]

Abstract. The paper presents an overview of the parallel computing models, architectures, and research projects that are based on asynchronous instruction scheduling. It starts with pure dataflow computing models and presents an historical development of several ideas (i.e. single-token-per-arc dataflow, tagged-token dataflow, explicit token store, threaded dataflow, large-grain dataflow, RISC dataflow, cycle-by-cycle interleaved multithreading, block interleaved multithreading, simultaneous multithreading) that resulted in modern multithreaded superscalar processors. The paper shows that unification of von Neumann and dataflow models is possible and preferred to treating them as two unrelated, orthogonal computing paradigms. Today's dataflow research incorporates more explicit notions of state into the architecture, and von Neumann models using many dataflow techniques to improve the latency hiding aspects of modern multithreaded systems.

Key words. parallel computer architectures, data-driven computing, multithread computing, static dataflow, tagged-token dataflow, threaded dataflow, cycle-by-cycle interleaving, block interleaving, multithreaded superscalar

AMS subject classifications. 68-02, 68M05, 68Q10

1. Introduction. There are many problems that require enormous computational capacity to solve, and therefore present opportunities for high-performance (parallel) computing. There are also a number of well-known hardware organization techniques for exploiting parallelism. In this paper we give an overview of the computers where parallelism is achieved by executing multiple asynchronous threads of instructions concurrently.

For sequential computers the principal execution model is the well known *von Neumann model* which consists of a sequential process running in a linear address space. The amount of concurrency available is relatively small [228]. Computers having more than one processor may be categorized in two types, SIMD or MIMD [82], and allow several *multiprocessor von Neumann* program execution models, such as *shared memory* (with actions coordinated by synchronization operations, e.g. P and V semaphore commands [75]), or *distributed memory* (with actions coordinated by message passing facilities [118, 135], e.g. explicit operating system calls). A MIMD computer in principle will have a different program running on every processor. This makes for an extremely complex programming environment. A frequent program execution model for MIMD computers is the *single-program multiple data* (SPMD) model [62] where the same program is run on all processors, but the execution may follow different paths through the program according to the processor's identity.

In multiprocessor systems, two main issues must be addressed: *memory latency*, which is the time that elapses between issuing a memory request and receiving the cor-

†Computer Systems Department, Jožef Stefan Institute, Jamova cesta 39, SI-1000 Ljubljana, Slovenia (`Jurij.Silc@ijs.si`).

‡Faculty of Computer and Information Science, University of Ljubljana, Tržaška cesta 25, SI-1000 Ljubljana, Slovenia, and Computer Systems Department, Jožef Stefan Institute, Jamova cesta 39, SI-1000 Ljubljana, Slovenia (`Borut.Robic@fri.uni-lj.si`).

§Department of Computer Design and Fault Tolerance, University of Karlsruhe, P.O.Box 6980, D-76128 Karlsruhe, Germany (`Ungerer@ira.uka.de`).

responding response, and *synchronization*, which is the need to enforce the ordering of instruction executions according to their data dependencies. The two issues cannot be properly resolved in a von Neumann context since connecting von Neumann processors into a very high speed general purpose computer also brings bottleneck problems [20].

As an alternative, the dataflow model was introduced as the radically new model capable of properly satisfying the two needs [8, 17, 66, 88]. Dataflow models use dataflow program graphs to represent the flow of data and control. Synchronization is achieved by a requirement that two or more specified events occur before an instruction is eligible for execution [191]. For example, in the *graph/heap model* [65] an instruction is enabled by the presence of simple tokens (i.e., data packets) on the arcs of copies of the dataflow graph, whereas the *unraveling interpreter model* (U-interpreter) [19] enables instructions by matching tags that identify the graph instance to which a token belongs. In [158] a detailed implementation of an unraveling dataflow interpreter of the Irvine Dataflow (Id) programming language is proposed. Dataflow models have also guided the design of several multithreaded computers [70].

Unfortunately, direct implementation of computers based on a pure dataflow model has been found to be an arduous task. For this reason, the impact of the convergence of the dataflow and control-flow was investigated [16, 125, 127, 141, 169]. In particular, *Nikhil* and *Arvind* posed the following question: *"What can a von Neumann processor borrow from dataflow to become more suitable for a multiprocessor?"* and offered the answer in terms of the P-RISC (Parallel RISC) programming model [164]. The model is based on a simple, RISC-like instruction set extended with three instructions that give a von Neumann processor a fine-grained dataflow capability [16]. It uses explicit *fork* and *join* commands. Based on P-RISC programming model a multithreaded Id programming language is implemented [162, 163]. Another model designed to support nested functional languages using sequential threads of instructions is the Multilisp model [114, 115, 116, 146]. Multilisp is an extension of the Lisp dialect Scheme with additional operators and semantics to deal with parallel execution. The principal language extension provided by Multilisp is $future(x)$. Upon executing $future(x)$, an immediate *undetermined* value is returned. The computation of x occurs in parallel and the result replaces *undetermined* when complete. Of course, any use of the result would block the parent process until the computation is finished.

The incorporation of conventional control-flow thread execution into the dataflow approach resulted in the *multithreaded* computer architecture which is one of the most promising and exciting avenues for the exploitation of parallelism [10, 43, 87, 128, 151, 159].

2. The day before yesterday: Pure dataflow. The fundamental principles of dataflow were developed by *Jack Dennis* [65] in the early 1970s. The dataflow model [8, 68, 93, 191] avoids the two features of von Neumann model, the program counter and the global updatable store, which become bottlenecks in exploiting parallelism [27]. The computational rule, also known as the *firing rule* of the dataflow model, specifies the condition for the execution of an instruction. The basic instruction firing rule, common to all dataflow systems is as follows: *An instruction is said to be executable when all the input operands that are necessary for its execution are available to it.* The instruction for which this condition is satisfied is said to be *fired*. The effect of firing an instruction is the consumption of its input values and generation of output values. Due to the above rule the model is *asynchronous*. It is also *self-scheduling*

since instruction sequencing is constrained only by data dependencies. Thus, the flow of control is the same as the flow of data among various instructions. As a result, a dataflow program can be represented as a directed *graph* consisting of named *nodes*, which represent instructions, and *arcs*, which represent data dependencies among instructions [64, 138] (Fig.2.1 a,b). Data values propagate along the arcs in the form of data packets, called *tokens*. The two important characteristics of the dataflow graphs are *functionality* and *composability*. Functionality means that evaluation of a graph is equivalent to evaluation of a mathematical function on the same input values. Composability means that graphs can be combined to form new graphs.

In a dataflow architecture the program execution is in terms of receiving, processing and sending out tokens containing some data and a tag. Dependencies between data are translated into tag matching and transformation, while processing occurs when a set of matched tokens arrives at the execution unit. The instruction which has to be fetched from the instruction store (according to the tag information) contains information about what to do with the data and how to transform the tags. The matching and execution unit are connected by an asynchronous pipeline, with queues added to smooth out load variations [136]. Some form of associate memory is required to support token matching. It can be a real memory with associative access, a simulated memory based on hashing, or a direct matched memory. Each solution has its proponent but none is absolutely suitable.

Due to its elegance and simplicity, the pure dataflow model has been the subject of many research efforts. Since the early 1970s, a number of dataflow computer prototypes have been built and evaluated, and different designs and compiling techniques have been simulated [17, 66, 88, 151, 168, 185, 196, 205, 208, 216, 219]. Clearly, an architecture supporting the execution of dataflow graphs should support the flow of data. Depending on the way of handling the data, several types of dataflow architectures emerged in the past [15]: *single-token-per-arc dataflow* [72], *tagged-token dataflow* [23, 229], and *explicit token store* [171]. We describe them in the following subsections.

2.1. Single-token-per-arc dataflow. The *single-token-per-arc* (or *static*) architecture was first proposed by *Dennis* [72] (Fig.2.1 c). At the machine level, a dataflow graph is represented as a collection of *activity templates*, each containing the *operation code* of the represented instruction, *operand slots* for holding operand values, and *destination address fields*, referring to the operand slots in subsequent activity templates that need to receive the result value. The static dataflow approach allows *at most one token* to reside on any one arc. This is accomplished by extending the basic firing rule as follows [67]: *A node is enabled as soon as tokens are present on its input arcs* and *there is no token on any of its output arcs.* To implement the restriction of at most one token per arc, *acknowledge signals*, traveling along additional arcs from consuming to producing nodes, are used as additional tokens. Thus, the firing rule can be changed to its original form.

The major advantage of the single-token-per-arc dataflow model is its simplified mechanism for detecting enabled nodes. Unfortunately, this model of dataflow has a number of serious drawbacks [15]. Since consecutive iterations of a loop can only partially overlap in time, only a pipelining effect can be achieved and thus a limited amount of parallelism can be exploited. Another undesirable effect is that token traffic is doubled. There is also a lack of support for programming constructs essential to any modern programming language. Despite these shortcomings, several machines were constructed and the most important are listed below. They made a

tremendous impact on the field of computer architecture by introducing radically new ways of thinking about massively parallel computers. They also served as a basis for subsequent dataflow computers.

MIT Static Dataflow Architecture: This machine was designed at the Massachusetts Institute of Technology (MIT) (Cambridge, Ma, USA) [66, 67, 69, 72] as a direct implementation of a single-token-per-arc dataflow model. It consisted of five major subsystems: a memory section, a processing section, an arbitration network, a control network, and a distribution network. All communication between subsystems was by asynchronous packet transmission over the channels. A prototype was built consisting of eight processing elements (emulated by microprogrammable microprocessors) and an equidistant packet routing network using 2×2 routing elements; it was used primarily as an engineering model [71].

LAU System: Researchers at the CERT (Toulouse, France) carried out the first research effort to go through the entire process of graph, language, and machine design. Their system, called LAU [49, 50, 175, 211], was built in 1979. "Language à assignation unique" (LAU) is a French acronym for single-assignment computation. The LAU system used static graphs and had its own dataflow language based on single-assignment.

TI's Distributed Data Processor: The Distributed Data Processor (DDP) [51, 52] was a system designed by Texas Instruments (USA) aimed to investigate the potential of dataflow as the basis of a high-performance computer. The DDP was not commercially exploited, but a follow-on design with Ada as the programming language [102] had application in military systems.

DDM1 Utah Data Driven Machine: DDM1 [63] was a computing element of a recursively structured dataflow architecture designed at the University of Utah (Salt Lake City, Ut, USA) and was completed at Burroughs Interactive Research Center (La Jolla, Ca, USA). The machine organization was based on the concept of recursion. It was tree structured, with each computing element connected to a superior element (above) and up to eight inferior elements (below). Each processing element consisted of an agenda queue, which contained firable instructions, an atomic memory (providing the program memory), and an atomic processor.

NEC Image Pipelined Processor: NEC Electronics (Japan) has developed the first dataflow VLSI microprocessor chip μPD7281 [47, 133, 134, 213, 214]. Its architecture was a pipeline with several blocks, or working areas, organized in a loop. A link table and function table stored object code, while the data memory was used for temporary storage of the tokens to be matched. The address generator and flow controller matched two tokens, which were temporarily stored in the queue before they were processed by the processing unit.

Hughes Dataflow Multiprocessor: The Hughes Dataflow Multiprocessor (HDFM) project began in 1981 at Hughes Aircraft Co. (USA), prompted by the need for high performance, reliable, and easily programmable processors for embedded systems. The HDFM [42, 89, 225, 226] consisted of many, relatively simple, identical processing elements connected by a global packet-switching network. The interconnection (3-D bussed cube) network was integrated with the processing element for ease of expansion and minimization of VLSI chip types. The communications network was designed to be reliable, with automatic retry on garble messages, distributed bus arbitration, alternate path packet routing, and failed processing element translation table to allow rapid switch-in and use of spare processing elements. The communication network was able to physically accommodate up to an $8 \times 8 \times 8$ configuration or

512 processing elements. Candidates proposed for the HDFM network were evaluated in [81]: a 3-D bussed cube network, a multi-stage network, a hypercube network, a mesh network, and a ring network. The processor engine was a three-stage pipelined processor with three operations overlapped: instruction fetch and dataflow firing rule check, instruction execution, and result and destination address combining to form a packet.

2.2. Tagged-token dataflow. The performance of a dataflow machine significantly increases when loop iterations and subprogram invocations can proceed in parallel. To achieve this, each iteration or subprogram invocation should be able to execute as a separate instance of a reentrant subgraph. However, this replication is only conceptual. In the implementation, only one copy of any dataflow graph is actually kept in memory. Each token includes a *tag*, consisting of the address of the instruction for which the particular data value is destined, and other information defining the computational context in which that value is used. This context is called the values *color*. Each arc can be viewed as a bag that may contain *an arbitrary* number of tokens with different tags. Now, the firing rule is: *A node is enabled as soon as tokens with identical tags are present at each of its input arcs.* Dataflow architectures that use this method are referred to as *tagged-token* (or *dynamic*) dataflow architectures (Fig.2.1 d). The model was proposed by *Watson* and *Gurd* at the University of Manchester (England) [229], and by *Arvind* and *Nikhil* at MIT [23].

The major advantage of the tagged-token dataflow model is the higher performance it obtains by allowing multiple tokens on an arc. One of the main problems of tagged-token dataflow model was the efficient implementation of the unit that collects tokens with matching colors. For reasons of performance, an associative memory would be ideal. Unfortunately, it would not be cost-effective since the amount of memory needed to store tokens waiting for a match tends to be very large. Therefore, all existing machines use some form of hashing techniques which are typically not fast enough compared with associative memory. In what follows, we list the most important tagged-token dataflow projects.

Manchester Dataflow Computer: The researchers at the University of Manchester focused on the construction of a prototype dataflow computer [28, 29, 38, 44, 45, 92, 107, 108, 109, 110, 111, 112, 129, 184, 218, 229, 230]. The graph structure in this machine was static with token labels to keep different procedure calls separate. The machine was first implemented as a simulation program in 1977/78, and as the hardware prototype in 1981. It consisted of a single processing element and a single structure storage module connected together via a simple 2×2 switch. The processing element had pipelined internal structure [59], with tokens passing through a queue module, a matching unit [60] and an instruction fetch unit before being processed by one of 20 function units in a parallel array that provided the computational power. The function units were microcoded, and different instructions took quite different times to execute. To overcome the restriction in the Manchester machine that only two successor instructions could be specified in one instruction, the TUPlicate operator (iterative instruction) was introduced [36, 37]. Using TUPlicate operator reduced the size of code and led to a significant reduction in execution time especially for large programs. The Extended Manchester Dataflow Computer (EXMAN) from Indian Institute of Science (Bangalore, India) [174] incorporated three major extensions to the basic machine: a multiple matching units scheme, an efficient implementation of the array data structure, and a facility to concurrently execute reentrant routines. To allow all storage functions to be performed concurrently also a prototype parallel

structure store was developed [139].

MIT Tagged-Token Dataflow Machine: This project originated at the University of California at Irvine (Ca, USA) and continued at MIT. The Irvine dataflow machine [21, 98, 99] implemented a version of the U-interpreter [97] and array handling mechanisms to support *I-structure* concepts [20, 23, 24, 157]. An I-structure may be viewed as a repository for data values obeying the single-assignment principle. That is, each element of the I-structure may be written only once but it may be read any number of times. The machine was proposed to consist of multiple clusters interconnected by a token ring. Each cluster (four processing elements [22]) shared a local memory through a local bus and a memory controller. The MIT machine [17, 90] was a modified Irvine machine, but still based on the Id language. Instead of using a token ring, an N × N packet switch network for interprocessor communications was used.

SIGMA-1: The SIGMA-1 [121, 122, 193, 232] system is a supercomputer for large-scale numerical computation and has been operational since early 1988 at the Electrotechnical Laboratory (Tsukuba, Japan). It consists of 128 processing elements and 128 structure elements interconnected by 32 local networks (10 × 10 crossbar packet switches) and one global two-stage Omega network. Sixteen maintenance processors are also connected with the structure elements and with a host computer for I/O operations, system monitoring, performance measurements, and maintenance operations [120].

PATTSY Processor Array Tagged-Token System: PATTSY [160] was an experimental system that had been developed at the University of Queensland (Brisbane, Australia), which supported the dynamic model of dataflow execution. PATTSY had a host computer that provided the user-interface to the machine and accepted user programs which it converted into dataflow graph. These graph were mapped onto the processing elements (PEs), but the actual scheduling of operation was carried out at runtime. An 18 processor prototype of the machine was operational. It used an IBM-PC as the host and the PEs were built from Intel 8085 based single board microcomputers.

NTT's Dataflow Processor Array System: The dataflow processors array system [212] from Nippon Telephone and Telegraph (Japan) was a dynamic tagged-token design intended for large scientific calculation. A hardware experimental system Eddy [13], consisting of 4 × 4 processing elements, was built and used to test some applications.

Q-p One-Chip Data-Driven Processor: The Q-p [25, 144, 166] is specifically designed to be a one-chip functional element which is easily programmable to form various dedicated processing functions. Particular design considerations were taken to achieve high flow-rate data-streams processing capabilities. In the Q-p, a novel bi-directional elastic pipeline processing concept was introduced to implement token matching. The Q-p was developed jointly by Osaka University, Sharp, Matsushita Electric Ind., Sanyo Electric, and Mitsubishi Electric (Japan).

DDDP Distributed Data Driven Processor: The DDDP [140] from OKI Electric Ind. (Japan) had a centralized tag manager and performed token matching using a hardware hashing mechanism similar to that of the Manchester Dataflow Computer. A prototype consisting of four processing elements and one structure store connected by a ring bus achieved 0.7 MIPS.

SDFA Stateless Data-Flow Architecture: SDFA was designed at the University of Manchester and inspired by the Manchester Dataflow Computer. As its name implies,

the SDFA system [204, 205] had no notion of states. There were no structure stores, and only extract-wait functionality was provided in the matching stores. All the instructions in the instruction-set were simple and based on RISC principles. There were no iterative or vector style instructions producing more than two tokens per execution.

CSIRAC II: The origins of the CSIRAC II dataflow computer date from 1978 [78]. It was built at the Swinburne Institute of Technology (Hawthorn, Australia). The architecture is unusual in that the temporal order of tokens with the same color on the same graph arc is maintained [76, 77].

PIM-D Parallel Inference Machine: The PIM-D was proposed to be one of the candidates for a parallel inference machine in the Fifth Generation Computer System and was a joint venture of ICOT and OKI Electric Ind. (Japan) [130, 131, 132]. This machine was constructed from multiple processing elements and multiple structure memories interconnected by a hierarchical network and exploited three types of parallelism: OR parallelism, AND parallelism, and parallelism in unification.

2.3. Explicit token store. One of the main problems of tagged-token dataflow architectures is the implementation of an efficient token matching. To eliminate the need for associative memory searches, the concept of an *explicitly address token store* has been proposed [171]. The basic idea is to allocate a separate memory frame for every active loop iteration and subprogram invocation. Each frame slot is used to hold an operand for a particular activity. Since access to frame slots is through offsets relative to a frame pointer, no associative search is necessary (Fig.2.1 e). To make this concept practical, the number of concurrently active loop iterations must be controlled. Hence, the constraint condition of *k-bounded loops* was proposed in [18, 53], allowing the number of concurrently active loop iterations to be bounded by a constant.

The explicit token address store principle was developed in the Monsoon project, but is applied in most dataflow architectures developed more recently (see Subsection 3.2. below), i.e. as so-called direct matching in the EM-4 and Epsilon-2.

Monsoon: The Monsoon dataflow multiprocessor was built jointly by MIT and Motorola (USA) [55, 172, 192, 217]. Monsoon dataflow processor nodes are coupled by a packet-communication network with each other and with I-structure storage units. The main objective of the Monsoon dataflow processor architecture was to alleviate the waiting/matching problem by explicitly using address token store. This is achieved by a dataflow processor using an eight-stage pipeline. The first pipeline stage is the instruction fetch stage which is arranged prior to the token matching, in contrast to dynamic dataflow processors with associative matching units. The new arrangement is necessary, since the operand fields in an instruction denote the offset in the memory frame that itself is addressed by the tag of a token. The explicit token address is computed by the composition of frame address and operand offset. This is done in the second stage, which is the first of three pipeline stages that perform the token matching. In the third stage a presence bit store access is performed to find out if the first token of a dyadic operation has already arrived. If that is not the case, the presence bit is set and the current token is stored of the frame slot in the frame store in the fourth stage. Otherwise the presence bit is reset and the operand retrieved from the frame store in the fourth pipeline stage. The next three stages are execution stages in the course of which the next tag is also computed concurrently. The eighth stage forms one or two new tokens that are sent to the network, stored in a user token queue, a system token queue, or directly recirculated to the instruction

fetch stage of the pipeline (see next section). Several small Monsoon systems have been built and installed at universities and research institutes.

(a)   *(b)*

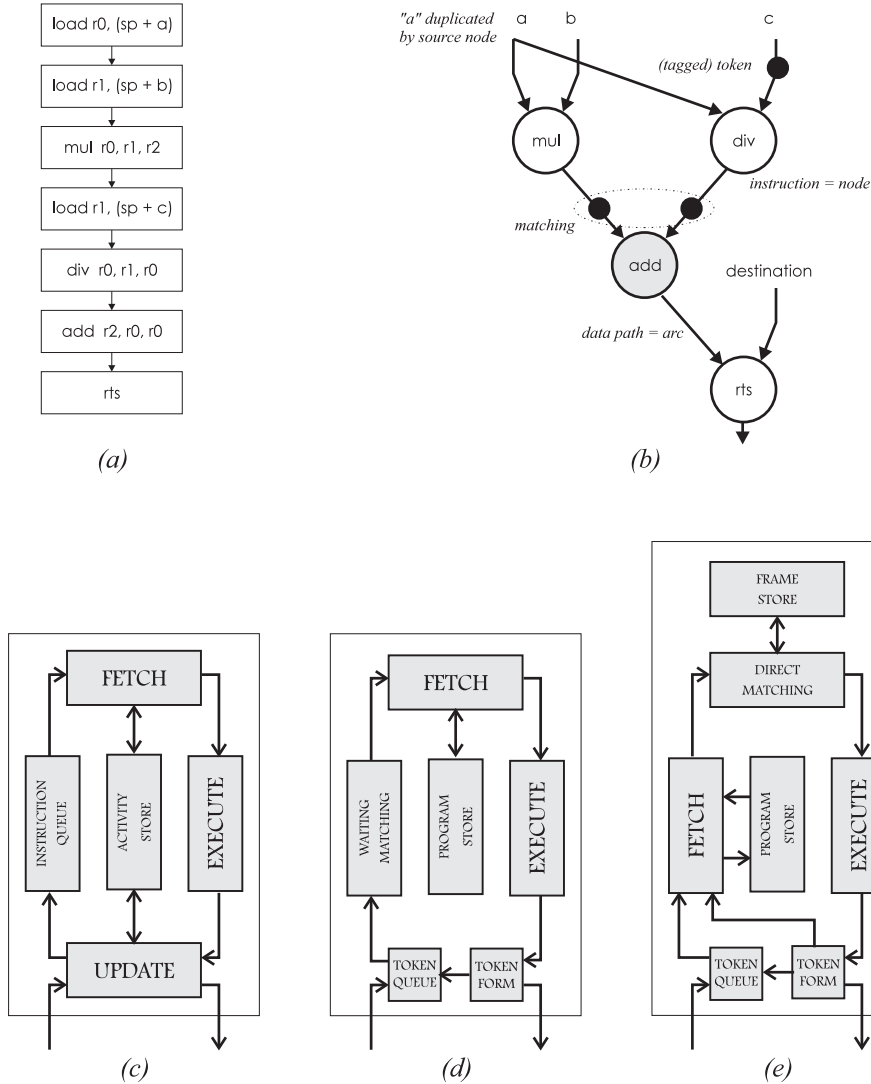*(c)*   *(d)*   *(e)*

Fig. 2.1 – Computing $a \cdot b + \frac{a}{c}$ with: (a) control flow; (b) dataflow. Pure dataflow basic execution pipeline: (c) single-token-per-arc dataflow; (d) tagged-token dataflow; (e) explicit token store dataflow.

3. Yesterday: Combining control-flow and dataflow. Pure dataflow computers based on the single-token-per-arc or tagged-token dataflow model usually perform quite poorly with sequential code. This is due to the fact that an instruction of the same thread can only be issued to the dataflow pipeline after the completion of its predecessor instruction. In the case of an 8-stage dataflow pipeline, an instruction of the same thread can be issued at most every eight cycles. If the load is low, for instance for a single sequential thread, the utilization of the dataflow processor

drops to one eighth of its maximum performance. A further drawback is the overhead associated with token matching. Before a dyadic instruction is issued to the execution stage, two result tokens are necessary. The first token is stored in the waiting-matching store, thereby introducing a bubble in the execution stage(s) of the dataflow processor pipeline. Only when the second token arrives is the instruction issued. For example, the pipeline bubbles sum up to 28.75 % executing a Traveling Salesman program on the Monsoon machine [172].

Since a context switch occurs in fine-grain dataflow after each instruction execution, no use of registers is possible. Use of registers could optimize the access time to data values, avoid pipeline bubbles caused by dyadic instructions, and reduce the total number of tokens during program execution.

One solution of these problems is combining the dataflow and control-flow mechanisms. The symbiosis between dataflow and von Neumann architectures is represented by a number of research projects developing von Neumann/dataflow hybrids [41, 127]. The spectrum of such hybrids is very broad, ranging from simple extensions of a von Neumann processor with a few additional instructions to specialized dataflow systems attempting to reduce overhead by increasing the execution grain size and employing various scheduling, allocation, and resource management techniques developed for von Neumann computers. These developments illustrate that dataflow and von Neumann computers do not necessarily represent two entirely disjoint worlds but rather are the two extreme ends of a spectrum of possible computer systems (Fig.3.1 a,b,c).

After *early hybrid dataflow* attempts, several techniques for combining control-flow and dataflow have emerged: *threaded dataflow* [173], *large-grain dataflow* [127], *RISC dataflow* [164], and *dataflow with complex machine operations* [80]. We describe them in the next five subsections (for a comparison of the techniques see [31]).

3.1. Early hybrid dataflow. The first attempts towards hybrid dataflow computing were made already in the early 1980s in quite diverse directions. Some of them are listed below.

JUMBO Newcastle Data-Control Flow Computer: The JUMBO [220] was built at the University of Newcastle upon Tyne (England) to study the integration of dataflow and control-flow computation. It has a packet communication organization with token matching. There are three principal units (matching unit, memory unit, processing unit) interconnected into ring by FIFO buffers.

MADAME Macro Dataflow Machine: The MADAME architecture from Jožef Stefan Institute (Ljubljana, Slovenia) was suitable for execution of acyclic dataflow (sub)graphs and supported synchronous dataflow computing [197, 198, 199, 200]. The advantage of such computing over pure dataflow is that more efficient runtime code can be generated because instructions can be scheduled at compile-time rather than at runtime.

PDF Piecewise Dataflow Architecture: The PDF architecture addressed Lawrence Livermore National Laboratory's (Livermore, Ca, USA) needs for supercomputing power. This architecture [177, 178] blended the strengths found in SIMD, MIMD, and dataflow architectures. The PDF contained a SIMD processing unit and a scalar processing unit that managed a group of processors (MIMD). Programs that ran on the PDF were broken into basic blocks and scheduled for execution using dataflow techniques. Two levels of scheduling divided responsibility between the software and hardware. The time-consuming analysis of when blocks can overlap was done in the compile-time, before program executions began. Individual instruction scheduling was done in hardware.

MUSE Multiple Stream Evaluator: The MUSE machine [40] from University of Nottingham (England) was a structured architecture supporting both serial and parallel processing which allowed the abstract structure of a program to be mapped onto the machine in a logical way. The MUSE machine had many features of dataflow architecture but in its detailed operation it steered a middle course between a pure dataflow machine, a tree machine, and the graph reduction approach.

RAMPS Real Time Acquisition and Multiprocessing System: RAMPS [30] was a parallel processing system developed at EIP Inc. (USA) for high performance data acquisition and processing applications. RAMPS used dataflow at a macro level while tasks were executed using a sequential model of computation.

DTN Dataflow Computer: The DTN Dataflow Computer [227] (developed by the Dutch company Dataflow Technology Nederland) is a high-performance workstation well suited for applications containing relatively small computing-intensive parts with enough parallelism to execute efficiently on the dataflow engine. The DTN Dataflow Computer contains a standard general purpose host, a graphical subsystem (four microprogrammable systolic arrays), and a dataflow engine. The dataflow engine consists of eight identical modules connected by a token routing network. Each module contains four NEC Image Pipelined Processors, interface chip, and image memory.

ETCA Functional Dataflow Architecture: A functional dataflow architecture has been developed at ETCA (Arcueil, France) and is dedicated to real-time processing [190]. Two types of data-driven processing elements, dedicated respectively to low and mid-level processing are integrated in a regular 3-D array. Its design relies on close integration of dataflow principles and functional programming. For the execution of low-level functions, a custom dataflow processor with six bi-directional I/O ports was developed [176].

3.2. Threaded dataflow. In a dataflow program each subgraph that exhibits a low degree of parallelism can be identified within a dataflow graph and transformed into a sequential thread. By the term *threaded dataflow* we understand a technique where the dataflow principle is modified so that instructions of a sequential instruction stream can be processed in succeeding machine cycles. A thread of instructions is issued consecutively by the matching unit without matching further tokens except for the first instruction of the thread. Threaded dataflow covers the repeat-on-input technique in the Epsilon-1 and Epsilon-2 processors, the strongly connected arc model of EM-4, and the direct recycling of tokens in Monsoon. Data passed between instructions from the same thread is stored in registers instead of writing them back to memory. These registers may be referenced by any succeeding instruction in the thread. Thereby single-thread performance is improved. The total number of tokens needed to schedule the instructions of a program is reduced thus saving hardware resources. Pipeline bubbles are avoided for dyadic instructions within a thread. Two threaded dataflow execution techniques can be distinguished: the *direct token recycling*, and *consecutive execution* of the instructions of a single thread. The first technique, used by the Monsoon dataflow computer, allows a particular thread to occupy only a single slot in the 8-stage pipeline, which implies that at least 8 threads must be active for a full pipeline utilization to be achieved. This cycle-by-cycle instruction interleaving of threads is used in a similar fashion by some multithreaded von Neumann computers (Section 4.). To optimize single-thread performance the Epsilon-processors and the EM-4 execute instructions from a thread consecutively. The circular pipeline of fine-grain dataflow is retained. However, the matching unit has to be enhanced with a mechanism that, after firing the first instruction of a thread, delays the matching

of further tokens in favor of issuing all instructions of the thread consecutively. By this mechanism cycling of tokens through the pipeline for the activation of the next instruction is suppressed. The three highly influential projects are given below.

Monsoon: The Monsoon dataflow processor can be viewed as a cycle-by-cycle interleaving multithreaded computer [173] due to its ability of direct token recycling. By use of this technique a successor token is directly fed back in the eight-stage Monsoon pipeline bypassing the token store. Another instruction of the same thread is executed every eighth processor cycle. Monsoon allows the use of registers (eight register sets are provided) to store intermediate results within a thread, thereby leaving the pure dataflow execution model.

Epsilon-2: Epsilon-2 machine [100, 101] developed at Sandia National Laboratories (Livermore, Ca, USA), supports a fully dynamic memory model, allowing single cycle context switches and dynamic parallelization. The system is built around a module consisting of a processor and structure unit, connected via a 4×4 crossbar to each other, an I/O port, and the global interconnect. The structure unit is used for storing data structures such as arrays, lists, and I-structures. The Epsilon-2 processor retains the high performance features of the Epsilon-1 prototype, including direct matching, pipelined processing, and a local feedback path. The ability to execute sequential code as a grain provides RISC-like execution efficiency.

EM-4 and EM-X: In the EM-4 project [35, 142, 181, 182, 183, 186, 187, 188, 189, 206] at Electrotechnical Laboratory (Tsukuba, Japan), the essential elements of a dynamic dataflow architecture using frame storage for local variables are incorporated into a single chip processor. In this design a *strongly connected* subgraph of a function body is implemented as a thread that uses registers for intermediate results. The EM-4 was designed for 1024 nodes. Since May 1990 the EM-4 prototype with 80 nodes (EMC-R gate-array processor) is operational. In 1993 the EM-X [143] (upgrade of EM-4) was designed to support latency reduction by fusing the communication pipeline with the execution pipeline, latency hiding via multithreading, and runtime latency minimization for remote memory access.

3.3. Large-grain dataflow. This technique, also referred to as *coarse-grain dataflow*, advocates activating macro dataflow actors by the dataflow principle while executing the represented sequences of instructions by the von Neumann principle [34, 61, 80, 127]. Large-grain dataflow machines typically decouple the matching stage (sometimes called signal stage, synchronization stage, etc.) from the execution stage by use of FIFO-buffers. Pipeline bubbles are avoided by the decoupling. Off-the-shelf microprocessors can be used for the execution stage. Most of the more recent dataflow architectures fall in this category and are listed below. Note, that they are often called *multithreaded machines* by their authors.

TAM Threaded Abstract Machine: The TAM [54, 56] from the University of California (Berkeley, Ca, USA) is an execution model for fine-grain interleaving of multiple threads, that is supported by an appropriate compiler strategy and program representation instead of elaborate hardware. TAM's key features are placing all synchronization, scheduling, and storage management under explicit compiler control.

*T: The *T (pronounced "start" [165]) is a direct descendant of dataflow architectures, especially of the Monsoon, and unifies them with von Neumann architectures. *T has a scalable computer architecture designed to support a broad variety of parallel programming styles including those which use multithreading based on non-blocking threads. A *T node consists of the data processor (executing threads), the remote-memory request coprocessor (for incoming remote load/store requests), and the syn-

chronization coprocessor (for handling returning load responses and join operations), which all share local memory. This hardware is coupled with a high performance network having a fat-tree topology with high cross-section bandwidth. A modified version of the *T [165] was implemented in cooperation by MIT and Motorola, based on a 88110 superscalar microprocessor (called 88110MP [32, 33]). Despite its name the StarT-ng [14, 46] bears no resemblance to its predecessor *T and to dataflow machines. It defines a rather conventional multiprocessor with PowerPC 620 processors augmented with special hardware for message-passing and shared memory access.

ADARC Associative Dataflow Architecture: In the Associative Dataflow Architecture ADARC [209, 210, 233] the processing units are connected via an associative communication network. The processors are equipped with private memories that contain instruction sequences generated at compile-time. The retrieval of executable instructions is replaced by the retrieval of input operands for the current instructions from the network. The structure of the network enables full parallel access to all previously generated results by all processors. A processor executes its current instruction (or instruction sequence) as soon as all requested input operands have been received [209]. In [233] it is shown that that ADARC is well-suited to implement neural network models. ADARC was developed at J.W.Goethe University (Frankfurt, Germany).

Pebbles: The Pebbles architecture [180] from Colorado State University (Fort Collins, Co, USA) is a large-grain dataflow architecture with a decoupling of the synchronization unit and the execution unit within the processing elements. The processing elements are coupled via a high-speed network. The local memory of each node consists of an instruction memory which is read by the execution unit and a data memory (or frame store) which is accessed by the synchronization unit. A ready queue contains the continuations representing those threads that are ready to execute. The frame store is designed as a storage hierarchy where a frame cache holds the frames of threads that will be executed soon. The execution unit is a 4-way superscalar microprocessor.

From Argument Fetch Dataflow Processor to MTA and EARTH: The EARTH (Efficient Architecture of Running Threads) [155], developed at McGill University and Concordia University (Montréal, Canada), is based on the MTA (Multithreaded Architecture) [86, 126] and dates back to the Argument Fetch Dataflow Processor. An MTA node consists of an execution unit that may be an off-the-shelf RISC microprocessor and a synchronization unit to support dataflow-like thread synchronization. The synchronization unit determines which threads are ready to be executed. Execution unit and synchronization unit share the processor local memory which is cached. Accessing data in a remote processor requires explicit request and send messages. The synchronization unit and execution unit communicate via FIFO queues: a ready queue containing ready thread identifiers links the synchronization unit with the execution unit, and an event queue holding local and remote synchronization signals connects the execution unit with the synchronization unit, but also receives signals from the network. A register use cache keeps track of which register set is assigned to which function activation [126]. MTA or EARTH rely on non-blocking threads. The EARTH architecture [161] is implemented on top of the experimental (but rather conventional) MANNA multiprocessor [91].

3.4. RISC dataflow. Another stimulus for dataflow/von Neumann hybrids was the development of *RISC dataflow* architectures, notably P-RISC [164], which allow the execution of existing software written for conventional processors. Using such a

machine as a bridge between existing systems and new dataflow supercomputers made the transition from imperative von Neumann languages to dataflow languages easier for the programmer.

P-RISC Parallel RISC: The basic philosophy underlying the development of the P-RISC architecture [164] can be characterized as follows: use a RISC-like instruction set, change the architecture to support multithreaded computation, add fork and join instructions to manage multiple threads, implement all global storage as I-structure storage, and implement the load/store instructions to execute in split-phase mode (Fig.3.1 d,e). The architecture based on the above principles was developed at MIT. It consists of a collection of PEs and Heap Memory Elements, interconnected through a packed-switching communication network.

3.5. Use of complex machine operations. Another technique to reduce the instruction level synchronization overhead is the use of complex machine instructions, for instance vector instructions. These instructions can be implemented by pipeline techniques as in vector computers. Structured data is referenced in block rather than element-wise, and can be supplied in a burst mode. This deviates from the I-structure scheme where each data element within a complex data structure is fetched individually from a structure store. A further advantage of complex machine operations is the ability to exploit parallelism at the subinstruction level. Therefore the machine has to partition the complex machine operation into suboperations that can be executed in parallel. The use of a complex machine operation may spare several nested loops. The use of a FIFO-buffer allows the machine to decouple the firing stage and the execution stage to bridge the different execution times within a mixed stream of simple and complex instructions issued to the execution stage. As a major difference to conventional dataflow architectures tokens do not carry data (except for the values "true" or "false"). Data is only moved and transformed within the execution stage. This technique is used in the Decoupled Graph/Computation Architecture, the Stollmann Dataflow Machine, and the ASTOR architecture. These architectures combine complex machine instructions with large-grain dataflow, described above. The structure-flow technique proposed for the SIGMA-1 enhances an otherwise fine-grain dataflow computer by structure load and structure store instructions, that move for instance whole vectors from or to the structure store. The arithmetic operations are executed by the cyclic pipeline within a single PE. A more detailed description is given below.

Decoupled Graph/Computation Architecture: In this architecture, developed at the University of Southern California (Los Angeles, Ca, USA), the token matching and token formatting and routing are reduced to a single graph operation called *determine executability* [80]. The decoupled graph/computation model separates the graph portion of the program from the computational portion. The two basic units of the decoupled model (*computational* unit and *graph* unit) operate in an asynchronous manner. The graph unit is responsible for determining executability by updating the dataflow graph, while the computation unit performs all the computational operations (fetch and execute).

Stollman Data Flow Machine: The Stollman dataflow machine [95, 96] from Stollman GmbH (Hamburg, Germany) is a coarse-grain dataflow architecture directed towards database applications. The dataflow mechanism is emulated on a shared-memory multiprocessor. The query tree of a relational query language (such as the database query language SQL) is viewed as dataflow graph. Complex database query instructions are implemented as coarse-grain dataflow instruction and (micro-)coded

as a traditional sequential program running on the emulator hardware.

<center>(a)   (b)   (c)</center>

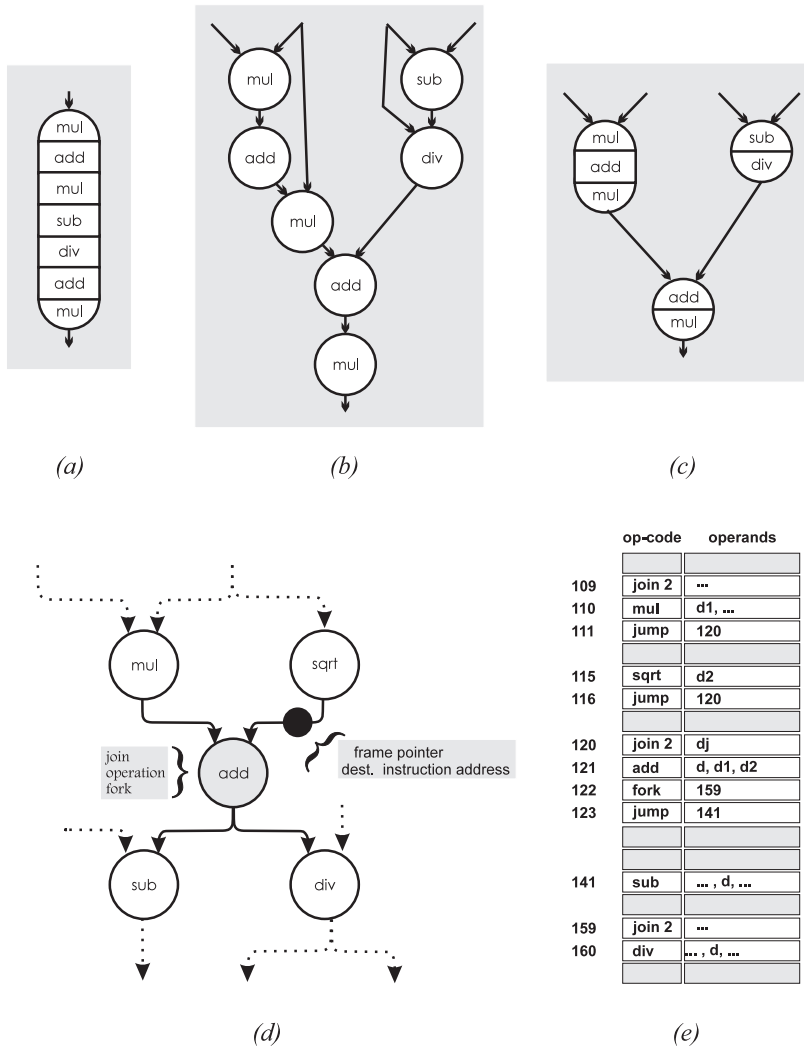<center>(d)   (e)</center>

Fig. 3.1 – Spectrum of scheduling models: (a) fully ordered (control flow); (b) partially ordered (dataflow); (c) partially ordered graph and fully ordered grains (e.g. Epsilon-2 hybrid). "RISCifying" dataflow (P-RISC dataflow/von Neumann hybrid): (d) conceptual; (e) encoding of graph.

ASTOR Augsburg Structure-Oriented Architecture: The ASTOR architecture [223, 224, 234, 235] from University of Augsburg (Germany) can be viewed as a dataflow architecture that utilizes task level parallelism by the architectural structure of a distributed memory multiprocessor, instruction level parallelism by a token-passing computation scheme, and subinstruction level parallelism by SIMD evaluation of complex machine instructions. Sequential threads of data instructions are compiled to dataflow macro actors and executed consecutively using registers. A dependency construct describes the partial order in the execution of instructions. It can be visualized by a dependency graph. The nodes in a dependency graph represent control

constructs or data instructions; the directed arcs denote control dependencies between the nodes. Tokens are propagated along the arcs of the dependency graph. To distinguish different activations of a dependency graph, a tag is assigned to each token. The firing rule of dynamic dataflow is applied: a node is enabled as soon as tokens with identical tags are present on all its input arcs. However, in the ASTOR architecture tokens do not carry data.

4. Today: Multithreading. In a *single-threaded* architecture the computation moves forward one step at a time through a sequence of states, each step corresponding to the execution of one enabled instruction. The state of a single-threaded machine consists of the *memory state* (program memory, data memory, stack) and the *processor state* which consists of *activity specifier* (program counter, stack pointer) and *register context* (a set of register contents). The activity specifier and the register context make up what is also called the *context* of a thread. Today most architectures are single-threaded architectures.

According to [70], a *multithreaded* architecture differs from the single-threaded architecture in that there may be several enabled instructions from different threads which all are candidates for execution (Fig.4.1). Similar to the single-threaded machine, the state of the multithreaded machine consists of the memory state and the processor state; the later, however, consists of a *collection* of activity specifiers and a *collection* of register contexts. A thread is a sequentially ordered block of instructions with a grain-size greater than one (to distinguish multithreaded architectures from fine-grained dataflow architectures).
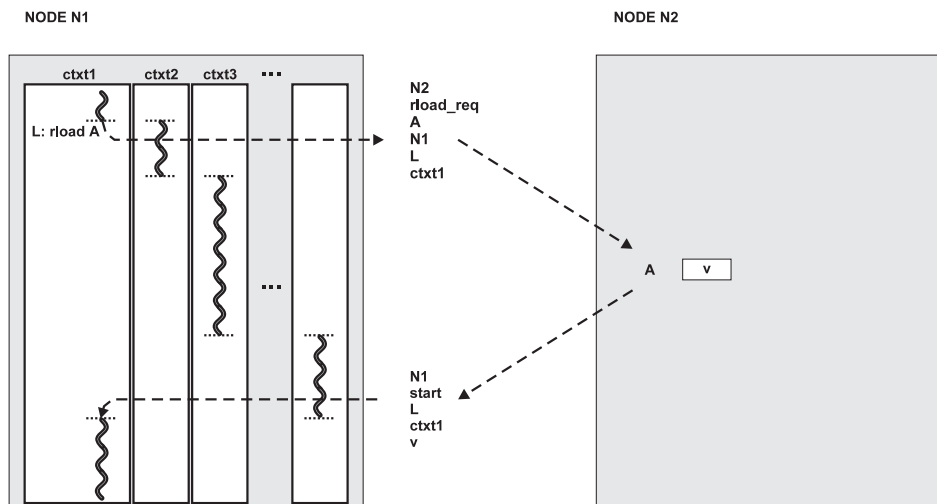

Fig. 4.1 – Multithreading: when a thread issues a remote-load request, the processor starts executing another thread.

Another notion is the distinction between *blocking* and *non-blocking* threads. A *non-blocking thread* is formed such that its evaluation proceeds without blocking the processor pipeline (for instance by remote memory accesses, cache misses or synchronization waits). Evaluation of a non-blocking thread starts as soon as all input operands are available, which is usually detected by some kind of dataflow principle. Thread switching is controlled by the compiler harnessing the idea of rescheduling rather than blocking when waiting for data. Access to remote data is organized

split-level by one thread sending the access request to memory and another thread activating when its data is available. Thus a program is compiled into many very small threads activating each other when data become available. The same hardware mechanisms may also be used to synchronize interprocess communications to awaiting threads, thereby alleviating operating systems overhead [159]. In contrast, a *blocking thread* might be blocked during execution caused by remote memory accesses, cache misses or synchronization needs. The waiting time, during which the pipeline is blocked, is lost when using a von Neumann processor, but can be bridged by a fast thread switch in a multithreaded processor. The thread is resumed when the reason for blocking is removed. Use of non-blocking threads typically leads to many small threads that are appropriate for execution by a hybrid dataflow computer or by a multithreaded architecture that is closely related to hybrid dataflow. Blocking threads may be normal threads or whole processes of a multithreaded UNIX-based operating system.

Notice, that we exclude in this section architectures that are designed for the execution of non-blocking threads. Although these architectures often are called *multithreaded*, we categorized them as threaded dataflow (Subsection 3.2.) or large-grain dataflow (Subsection 3.3.) since a dataflow principle is applied to start execution of non-blocking threads. Thus, multithreaded architectures (in the more narrow sense applied here) stem from the modification of RISC or even of superscalar RISC processors. They are able to bridge waiting times that arise in the execution of blocking threads by fast thread switching. This ability is in contrast to RISC processors or today's superscalar processors, that use busy waiting or a time-consuming, operating system based thread switch. One characteristic for such multithreaded architectures is the use of multiple register sets and a mechanism that dynamically triggers the thread switch. Thread switch overhead must be very low, from zero to only a few cycles.

Current microprocessors utilize instruction-level parallelism by a deep processor pipeline and by the superscalar instruction issue technique. A superscalar processor is able to dispatch multiple instructions each clock cycle from a conventional linear instruction stream [73]. DEC Alpha 21164, IBM & Motorola PowerPC 604 and 620, MIPS R10000, Sun UltraSparc and HP PA-8000 issue up to four instructions per cycle from a single thread. VLSI-technology will allow future generations of microprocessors to exploit instruction-level parallelism up to 8 instructions per cycle, or more.

However, the instruction-level parallelism found in a conventional instruction stream is limited. Recent studies showed the limits of processor utilization even of today's superscalar microprocessors. Using the SPEC92 benchmark suite, the PowerPC 620 showed an execution of 0.96 to 1.77 instructions per cycle [74], and even an 8-issue Alpha processor will fail to sustain 1.5 instructions per cycle [222].

The solution is the additional utilization of more coarse-grained parallelism. The main approaches are the *multiprocessor chip* and the *multithreaded processor*. The multiprocessor chip integrates two or more complete processors on a single chip. Therefore every unit of a processor is duplicated and used independently of its copies on the chip. In contrast, the multithreaded processor stores multiple contexts in different register sets on the chip. The functional units are multiplexed between the threads that are loaded in the register sets. Depending on the specific multithreaded processor design, only a single instruction pipeline is used, or a single dispatch unit issues instructions from different instruction buffers simultaneously. Because of the multiple register sets, context switching is very fast. Multithreaded processors tolerate

memory latencies by overlapping the long-latency operations of one thread with the execution of other threads - in contrast to the multiprocessor chip approach. While the multiprocessor chip is easier to implement, use of multithreading in addition to a wide issue bandwidth is a promising approach.

Research on multithreaded architectures has been motivated by two concerns: tolerating latency and bridging of synchronization waits by rapid context switches. Three different approaches of multithreaded architectures are distinguished: *cycle-by-cycle interleaving* [117, 173, 202, 215], *block interleaving* [2, 3, 106], and *simultaneous multithreading* [221, 222].

4.1. Cycle-by-cycle interleaving. In the *cycle-by-cycle* interleaving model the processor switches to a different thread after each instruction. In principle, an instruction of the same thread is fed in the pipeline after the completion of the execution of the previous instruction. There must be at least as many register sets (loaded threads) available on the processor as the number of pipeline stages. Since cycle-by-cycle interleaving eliminates control and data dependencies between instructions in the pipeline, pipeline conflicts cannot arise and the processor pipeline can be easily built without the necessity of complex forwarding paths. This leads to a very simple and therefore potentially very fast pipeline - no hardware interlocking is necessary. The latency is tolerated by not scheduling a thread until its reference to a remote memory has completed. This model requires a large number of threads and complex hardware to support them. Interleaving the instructions from many threads also limits the processing power accessible to a single thread, thereby degrading the single-thread performance. The use of the dependence look-ahead technique in HEP, Horizon and Tera gives some help. Some machines that use cycle-by-cycle interleaving are given below.

HEP Heterogeneous Element Processor: The HEP system was a MIMD shared-memory multiprocessor system developed by Denelcor Inc. (Denver, Co, USA) between 1978 and 1985 [137, 145, 201, 202, 203], and was a pioneering example of a multithreaded machine. Spatial switching occurred between two queues of processes, one of these controlled program memory, register memory, and the functional memory while the other controlled data memory. The main processor pipeline had eight stages, matching the number of processor cycles necessary to fetch a data item from memory in register. Consequently eight threads were in execution concurrently within a single HEP processor. In contrast to all other cycle-by-cycle interleaving processor, all threads within a HEP processor shared the same register set. Multiple processors and data memories were interconnected via a pipelined switch and any register memory or data memory location could be used to synchronize two processes on a producer-consumer basis by a full/empty bit synchronization on a data memory word.

MASA Multilisp Architecture for Symbolic Applications: The MASA was a multithreaded processor architecture for parallel symbolic computation with various features intended for effective Multilisp program execution [85, 117]. MASA featured a tagged architecture, multiple contexts, fast trap handling, and a synchronization bit in every memory word. Its principal novelty was the use of multiple contexts both to support interleaved execution from separate instruction streams and to speed up procedure calls and trap handling (in the same manner as register windows).

Horizon: This (paper) architecture was based on the HEP but extended to a massively parallel MIMD computer. The machine was designed for up to 256 processing elements and up to 512 memory modules in a 16 × 16 × 6 node internal network. Like HEP it employed a global address space, and memory-based synchronization

through the use of full/empty bits at each location [94, 148, 215]. Each processor supported 128 independent instruction streams by 128 register sets with context switches occurring at every clock cycle.

Tera MTA Multi-Threaded Architecture: The Tera MTA [9, 11, 12] is based on the Horizon architecture and currently under construction by Tera Computer Company (Seattle, Wa, USA) (after many delays introduction to market is scheduled for Spring 1997). The machine is a vast multiprocessor with multithreaded processor nodes arranged in 3D mesh of pipelined packet-switch nodes. In the case of the maximum configuration of 4096 nodes, arranged in a $16 \times 16 \times 16$ mesh, there are up to 256 processors, 512 memory units, 256 I/O cache units, 256 I/O processors, and 2816 switching nodes. Each processor is 64-bit custom chip with up to 128 simultaneous threads in execution. It alternates between ready threads, using a deep pipeline. Inter-instruction dependencies are explicitly encoded by the compiler. Each thread has a complete set of registers. Memory units have 4-bit tags on each word, or full/empty and trap bits [12]. The Tera MTA exploits parallelism at all levels, from fine-grained instruction-level parallelism within a single processor to parallel programming across processors, to multiprogramming among several applications simultaneously. Consequently, processor scheduling occurs at many levels, and managing these levels poses unique and challenging scheduling concerns [11].

SB-PRAM and HPP: The SB-PRAM [1, 26] or HPP (High Performance PRAM) [83, 84] is a MIMD parallel computer with shared address space and uniform memory access time due to its motivation: building a multiprocessor that is as close as possible to the theoretical machine model CRCW-PRAM. Processor and memory modules are connected by a butterfly network. Network latency is hidden by pipelining several so-called virtual processors on one physical processor node in cycle-by-cycle interleaving mode. A first prototype (SB-PRAM) [26] is running with four processors, a second prototype (HPP) is under construction. In the SB-PRAM 32 virtual processors are scheduled round-robin for every instruction [83]. The project is running at the University of Saarland (Saarbrücken, Germany).

4.2. Block interleaving. The block-interleaving approach, exemplified by the MIT Sparcle processor, executes a single thread until it reaches a long-latency operation, such as a remote cache miss or a failed synchronization, at which point it switches to another context. The Rhamma processor switches contexts whenever a load, store or synchronization operation is discovered.

In the *block* interleaving model a thread is executed for many cycles before context switching. Context switches are used only to hide long memory latencies since small pipeline delays are hidden by proper ordering of instructions performed by the optimizing compiler. Since multithreading is not used to hide pipeline delays, fewer total threads are needed and a single thread can execute at full speed until the next context switch. This may also simplify the hardware. There are three variations of this model. The *switch-on-load* version switches only on instructions that load data from shared memory while storing data in shared memory does not cause context switching (since local memory loads and other instructions all complete quickly and can be scheduled by the compiler). However, context switches sometimes occur sooner than needed: If a compiler ordered instructions so that a load from shared memory was issued several cycles before the value was used, the context switch should not have to occur until the actual use of the value. This strategy is implemented in the *switch-on-use* model [154]. Here, a *valid bit* is added to each register. The bit is cleared when the loading from shared memory to the corresponding register is issued and set when the result

returns from the network. A thread context switches if it needs a value from a register whose valid bit is still cleared. A benefit of this model is that several load instructions can be grouped together thus prefetching several operands of an instruction. Instead of using valid bits, an explicit context switch instruction can be added between the group of loads and their subsequent uses. This model, which is called *explicit-switch* [56, 165], is simpler to implement and requires only one additional instruction.

The multithreading with caching approach adds *caches* to the block interleaving model. Only those loads that miss in the cache have long latencies and cause context switches. By filtering out many of the remote references, caching reduces the number of threads. In [39], it is shown that for most applications just two or three threads per processor is sufficient. There are three variations of the cache-extended block interleaving model. The *switch-on-miss* model [5] context switches if load from shared memory instructions misses in the cache. Such a context switch is not detected immediately, however, so a number of subsequent instructions have already entered the CPU pipeline and thus wasted CPU time. The *switch-on-use-miss* model [106] context switches when an instruction tries to use the (still missing) value from a shared load that missed in the cache. The *conditional-switch* model provides the benefits of grouping (of the explicit-switch model) and caching. In this model, the explicit switch instruction is ignored if all load instructions (in the preceding group) hit the cache; otherwise, the context switch is performed. In what follows, we describe five representatives of the block interleaving model of multithreading.

CHoPP Columbia Homogeneous Parallel Processor: The CHoPP [154] was a shared memory MIMD with up to 16 powerful computing nodes. High sequential performance is due to issuing multiple instructions on each clock cycle, zero-delay branch instructions, and fast execution of individual instructions. Each node can support up to 64 threads.

Sparcle Processor in Alewife System: The MIT Alewife multiprocessor [3, 4, 5, 6, 7, 147] is based on the multithreaded Sparcle processor. The Sparcle processor is derived from a Sparc RISC processor. The eight overlapping register windows of a Sparc processor are organized in four independent non-overlapping thread contexts, each using two windows (one as register set, the other as a context for trap and message handlers). Thread switching is triggered by external hardware when a remote memory access is detected. Emptying the pipeline from instructions of the thread that caused the context switch and organizational software overhead sum up to a context switching penalty of 14 processor cycles. The Alewife multiprocessor has been operational since May 1994 [4]. It uses a low-dimensional direct interconnection network. Despite its distributed-memory architecture, Alewife allows efficient shared-memory programming through a multilayered approach to locality management. Communication latency and network bandwidth requirements are reduced by a directory-based cache-coherence scheme referred to as LimitLESS directories. Latencies still occur although communication locality is enhanced by runtime and compile-time partitioning and placement of data and processes.

MSparc: An approach similar to the Sparcle processor is taken at the University of Oldenburg (Germany) with the MSparc processor [156, 179]. MSparc supports up to four contexts on chip and is compatible to standard Sparc processors. Switching is supported by hardware and can be achieved within one processor cycle. The multithreading policy is block interleaving with the switch-on-cache-miss policy as in the Sparcle processor.

J-Machine: The MIT Jellybean Machine [58, 167] is so called because it is to

be built entirely of a large number of "jellybean" components. The initial version uses an 8 × 8 × 16 cube network, with possibilities of expanding to 64K nodes. The "jellybeans" are message driven processor (MDP) chips, each of which has a 36-bit processor, a 4K word memory, and a router with communication ports for 6 directions. External memory of up to 1 M words can be added per processor. The MDP creates a task for each arriving message. In the prototype, each MDP chip has 4 external memory chips that provide 256K memory words. However, access is through a 12-bit data bus, and with an error correcting cycle, the access time is four memory cycles per word. Each communication port has a 9-bit data channel. The routers provide support for automatic routing from source to destination. The latency of a transfer is 2 microseconds across the prototype machine, assuming no blocking. When a message arrives, a task is created automatically to handle it in 1 microsecond. Thus, it is possible to implement a shared memory model using message passing, in which a message provides a fetch address and an automatic task sends a reply with the desired data.

Rhamma: The multithreaded Rhamma processor [103, 104] from the University of Karlsruhe (Germany) uses a fast context switch to bridge latencies caused by memory accesses or by synchronization operations. Load/store, synchronization and execution operations of different threads are executed simultaneously by specialized functional units within the processor. The units are coupled by FIFO buffers and access different register sets. Each unit stops the execution of a thread when it recognizes an instruction intended for another unit. To perform a context switch the unit passes the thread tag to the FIFO buffer of the unit that is appropriate for the execution of the instruction. Then the unit resumes processing with another thread of its own FIFO buffer. The Rhamma processor is most similar to the Sparcle. However, the execution unit of the Rhamma processor switches the context whenever it comes across a load, store or synchronization instruction, and the load/store unit switches whenever it meets an execution or synchronization instruction (switch-on-load policy). In contrast to Sparcle, the context switch is in an early stage of the pipeline, thus decreasing context switching time. On the other hand, the overall performance of the Rhamma processor suffers from the higher rate of context switches unless the context switch time is very small. Software simulations showed the need for extremely fast context switching. Therefore a VHDL representation of Rhamma with optimizations towards fast context switching was developed, simulated and synthesized. Specific implementation techniques reduce switching costs to zero or at most one processor cycle. These techniques use a context switch buffer which is a table containing the addresses of instructions that already yielded a context switch.

4.3. Simultaneous multithreading or multithreaded superscalar. The *simultaneous multithreading* or *multithreaded superscalar* approach [152, 194, 195, 221, 222] combines a wide issue superscalar instruction dispatch with the multiple context approach by providing several register sets on the multiprocessor and issuing instructions from several instruction queues simultaneously. Therefore, the issue slots of a wide issue processor can be filled by operations of several threads. Latencies occurring in the execution of single threads are bridged by issuing operations of the remaining threads loaded on the processor. In principle, the full issue bandwidth can be utilized. Project dependent details are briefly discussed below.

Media Research Laboratory Processor: The multithreaded processor of the Media Research Laboratory of Matsushita Electric Ind. (Japan) [123, 124] is the first approach to simultaneous multithreading. Instructions of different threads are is-

sued simultaneously to multiple functional units. Simulation results on a parallel ray-tracing application showed that using 8 threads a speed-up of 3.22 in case of one load/store unit and of 5.79 in case of two load/store units can be achieved over a conventional RISC processor. However, caches or TLBs are not simulated, nor is a branch prediction mechanism.

Simultaneous Multithreading: The *simultaneous multithreading* approach from the University of Washington (Seattle, Wa, USA) [79] surveys enhancements of the Alpha 21164 processor [222] and of a hypothetical out-of-order issue superscalar microprocessor that resembles R10000 and PA-8000 [221]. Simulations were conducted to evaluate processor configurations of an up to 8-threaded and 8-issue superscalar. This maximum configuration showed a throughput of 6.64 instructions per cycle due to multithreading using the SPEC92 benchmark suite and assuming a processor with 32 functional units (among them multiple load/store units) [222]. The second approach [221] evaluated more realistic processor configurations and reached a throughput of 5.4 instructions per cycle for the 8-threaded and 8-issue superscalar case. Implementation issues and solutions to register file access and instruction scheduling were also presented.

Multithreaded Superscalar (Sigmund & Ungerer): While the simultaneous multithreading approach [222] surveys enhancements of the Alpha 21164 processor, the *multithreaded superscalar* processor approach from the University of Karlsruhe [194, 195] is based on a simplified PowerPC 604 processor. Both approaches, however, are similar in their instruction issuing policy. The multithreaded superscalar processor implements the six-stage instruction pipeline of the PowerPC 604 processor (fetch, decode, dispatch, execute, complete, and write-back) [207] and extends it to employ multithreading. The processor uses various kinds of modern microarchitecture techniques as e.g. separate code and data caches, branch target address cache, static branch prediction, in-order dispatch, independent execution units with reservation stations, rename registers, out-of-order execution, and in-order completion. However the instruction set is simplified (using an extended DLX [119] instruction set instead). A software simulation evaluated various configurations of the multithreaded superscalar processor. While the single-threaded 8-issue superscalar processor only reached a throughput of about 1.14 instructions per cycle, the 8-threaded 8-issue superscalar processor executed 4.19 instructions per cycle (the load/store frequency in the work load sets the theoretical maximum to 5 instructions per cycle). Increasing the superscalar issue bandwidth from 4 to 8 yields only a marginal gain in instruction throughput - a result that is nearly independent of the number of threads in the multithreaded processor. A multiprocessor chip approach with 8 single-threaded scalar processors reaches 6.07 instructions per cycle. Using the same number of threads, the multiprocessor chip reaches a higher throughput than the multithreaded superscalar approach. However, if the chip costs are taken into consideration, a 4-threaded 4-issue superscalar processor outperforms a multiprocessor chip built from single-threaded processors by a factor of 1.8 in performance/cost relation [194].

Multithreaded Superscalar (Bagherzadeh et al.): This *multithreaded superscalar* processor approach [105, 153] combines out-of-order execution within an instruction stream with the simultaneous execution of instructions of different instruction streams. A particular superscalar processor called Superscalar Digital Signal Processor (SDSP) is enhanced to run multiple threads. The enhancements are directed by the aim of minimal modification to the superscalar base processor. Therefore most resources on the chip are shared by the threads, as for instance the register file, reorder buffer,

instruction window, store buffer, and renaming hardware. Based on simulations a performance gain of 20 - 55 % due to multithreading was achieved across a range of benchmarks. The project runs at the University of California at Irvine.

5. Coming: Micro dataflow and nanothreading. The latest generation of microprocessors - as exemplified by the Intel PentiumPro [48], MIPS R10000 [231], and HP PA-8000 [149] - displays an out-of-order dynamic execution that is referred to as *local dataflow* or *micro dataflow* by microprocessor researchers.

In the first paper on the PentiumPro, the instruction pipeline is described as [48]: *"The flow of the Intel Architecture instructions is predicted and these instructions are decoded into micro-operations (micro-ops), or series of micro-ops, and these micro-ops are register-renamed, placed into an out-of-order speculative pool of pending operations, executed in dataflow order (when operands are ready), and retired to permanent machine state in source program order."* After register renaming the instructions (or micro-ops) are placed in an instruction window of pending instructions and in a reorder buffer that saves the program order and execution states of the instructions (instruction window and reorder buffer may coincide). State-of-the art microprocessors typically provide 32 (R10000), 40 (PentiumPro) or 56 (PA-8000) instruction slots in the instruction window or reorder buffer. Instructions are ready to be executed as soon as all operands are available. A 4-issue superscalar processor dispatches up to 4 instructions per cycle to the execution units provided that resouce conflicts do not occur. Dispatch and execution determine the out-of-order section of a microprocessor. After execution instructions are retired in program order.

Comparing dataflow computers with such superscalar microprocessors shows several similarities and differences which are briefly discussed below.

While a single thread of control in modern microprocessors often does not incorporate enough fine-grained parallelism to feed the multiple functional units of todays microprocessors, dataflow resolves any threads of control into separate instructions that are ready to execute as soon as all required operands are available. Thus, the fine-grained parallelism potentially utilized by a dataflow computer is far larger than the parallelism available for microprocessors.

Data and control dependencies potentially cause pipeline hazards in microprocesssors that are handled by complex forwarding logic. Due to the continuous context switches in fine-grain dataflow computers and in cycle-by-cycle interleaving machines, pipeline hazards are avoided with the disadvantage of a poor single thread performance.

Antidependencies and output dependencies are removed by register renaming that maps the architectural registers to the physical registers of the microprocessor. Thereby the microprocessor internally generates an instruction stream that satisfies the single assignment rule of dataflow. Modern microprocessors remove antidependencies and output dependencies on-the-fly and avoid the high memory requirements, the often awkward solutions for data structure storage and manipulation, and for loop control caused by the single assignment rule in dataflow computers.

The main difference between the dependence graphs of dataflow and the code sequence in an instruction window of a microprocessors is branch prediction and speculatively execution. The accuracy of the branch prediction is surprisingly high - more than 95 % are reported for single SPECmark programs. However, rerolling execution in case of a wrongly predicted path, is costly in processor cycles especially in deeply pipelined microprocessors. The idea of branch prediction and speculative execution has never been evaluated in the dataflow environment.

Due to the single thread of control, a high degree of data and instruction locality is present in the machine code of a microprocessor. The locality allows to employ a storage hierarchy that stores the instructions and data potentially executed in the next cycles close to the executing processor. Due to the lack of locality in a dataflow graph, a storage hierarchy is difficult to apply in dataflow computers.

The matching of executable instructions in the instruction window of microprocessors is restricted to a part of the instruction sequence. Because of the serial program order, the instructions in this window are likely to be executable soon. Therefore the matching hardware can be restricted to a small number of instruction slots. In dataflow computers the number of tokens waiting for a match can be very high. A large waiting-matching store is required. Due to the lack of locality the likelyhood of the arrival of a matching token is difficult to estimate so caching of tokens to be matched soon is difficult in dataflow.

An unsolved problem in todays microprocessors is the memory latency caused by cache misses. As reported for the SGI Origin 2000 distributed shared memory system [150] these latencies are 11 processor cycles for a primary-level cache miss, 60 cycles for a secondary-level cache miss, and can be up to 180 cycles for a remote memory access. In number of missed instruction slots the latencies should be multiplied by the degree of superscalar. Only a small part of the memory latency can be removed or hidden in modern microprocessors even when techniques like out-of-order execution, write buffer, cache preload hardware, lockup free caches, and a pipelined system bus are employed. So microprocessors often idle and are unable to exploit the high degree of internal parallelism provided by a wide superscalar approach. The rapid context switching of dataflow and multithreaded architectures shows a superior way out. Idling is avoided by switching execution to another context.

An 8-issue (or even higher) superscalar dispatch will be possible in the next generation of microprocessors. Finding enough fine-grain parallelism to fully exploit the processor will be the main problem. One solution is to *enlarge the instruction window* to several hundred instruction slots with hopefully more simultaneously executable instructions present. There are two draw-backs to this approach. First, regarding that all instructions stem from a single instruction stream and that on average every seventh instruction is a branch instruction, most of the instructions in the window will be speculatively assigned with a very deep speculation level (todays depth normally is four at maximum). Thereby most of the instruction execution will be speculative. The principal problem here arises from the single instruction stream that feeds the instruction window. Second, if the instruction window is enlarged, the updating of the instruction states in the slots and matching of executable instructions lead to more complex hardware logic in the dispatch stage of the pipeline thus limiting the cycle rate increase which is essential for next generations of microprocessors. Solutions are the *decoupling of the instruction window* with respect to different instruction classes as in the PA-8000, the *partitioning of the dispatch stage* into several pipeline stages, and *alternative instruction window organizations*. One alternative instruction window organization is the multiple FIFO-based organization in the dependence-based microprocessor [170]. Only the instructions at the heads of a number of FIFO buffers can be dispatched to the execution units in the next cycle. The total parallelism in the instruction window is restricted in favour of a less costly dispatch that does not slow down processor cycle rate. Thereby the potential fine-grained parallelism is limited - a technique somewhat similar to the threaded dataflow approaches described above. It might be interesting to look, with respect to alternative instruction window or-

ganizations, at dataflow matching store implementations and dataflow solutions like threaded dataflow as exemplified by the repeat-on-input technique in the Epsilon-2 and strongly-connected arcs model of EM-4, or the associative switching network in the ADARC, etc. For example, the repeat-on-input strategy dispatches very small compiler-generated code sequences serially (in an otherwise fine-grained dataflow computer). Transfered to the local dataflow in an instruction window, a dispatch string might be used where a serie of data dependent instructions are generated by a compiler and dispatched serially after the dispatch of the leading instruction. However, the high number of speculative instructions in the instruction window remains.

Another new idea is the *nanothreading* technique of DanSoft (Košice, Slovak Republic) [113]. Nanothreading dismisses full multithreading for a nanothread that executes in the same register set as the main thread. The DanSoft nanothread requires only a 9 bit program counter, some simple control logic, and it resides in the same page as the main thread. The nanothread might be used to fill the instruction dispatch slots of a wide superscalar approach (like in simultaneous multithreading) without the need for several register sets.

But the main problem of todays superscalar microprocessor may not be found in the instruction window organization. It is the necessity of instruction completion/retirement due to the serial semantic of the instruction stream. For example, if a load instruction causes a secondary-level cache miss, the whole reorder buffer may soon be clogged by succeeding instructions (succeeding in sequential program order) that are already executed. Because of the sequential program order that should be restored by the completion stage, these instructions cannot be retired and removed from the reorder buffer even if the instructions are independent of the load instruction that caused the cache miss. The completion may be the main obstacle for further performance increase in microprocessors.

In principle, an algorithm defines a partial ordering of instructions due to control and true data dependences. The total ordering in an instruction stream for todays microprocessors stems from von Neumann languages. Why should

- a programmer design a partially ordered algorithm, and
- code the algorithm in total ordering because of the use of a sequential von Neumann language,
- the compiler regenerate the partial order in a dependence graph, and
- generate a reordered "optimized" sequential machine code,
- the microprocessor dynamically regenerate the partial order in its out-of-order section, execute due to a *micro dataflow* principle,
- and then reestablish the unnatural serial program order in the completion stage ?

Ideally, an algorithm should be coded in an appropriate higher-order language (e.g. dataflow-like languages might be appropriate). Next, the compiler should generate machine code that still reflects the parallelism and not an unnecessary serialization. Here, a dataflow graph viewed as machine language might show the right direction. A parallelizing compiler may generate this kind of machine code even from a program written in a sequential von Neumann language. The compiler could use compiler optimization and coding to simplify the dynamic analysis and dispatch out of the instruction window. The processor dismisses the serial reordering in the completion stage in favour of an only partial reordering. The completion unit retires instructions not in a single serial order but in two or more series (as in the simultaneous multithreaded processors). Clogging of the reorder buffer is avoided since clogging of a

thread does not restrict retirement of instructions of another thread.

6. Conclusion. Conceptually, the research in parallel computing shows a clear unification of two paradigms, i.e. von Neumann and dataflow computing. One of the key realizations made by the dataflow community is that pure dataflow and pure control-flow are not orthogonal to each other but are at two ends of a "continuum." Hence, the trend in dataflow research was to incorporate explicit notions of state into architecture. As a result, many of the pure dataflow ideas were replaced by abstractions that are more conventional. At the same time, von Neumann computing incorporated many of the dataflow techniques. Based on these and other merging ideas the multithreaded model of computing emerged within the "continuum" with several theoretically and commercially fruitful advantages.

Historically, commercial parallel machines have demonstrated innovative organizational structures, often tied to a particular programming model, as architects sought to obtain the ultimate in performance out of any given technology. The period up to 1985 is dominated by advancements in *bit-level parallelism* (8-bit microprocessors was replaced by 16-bit, 32-bit etc.). Doubling the width of the datapath reduces the number of cycles required to perform an operation. The period from the mid-80s to mid-90s is dominated by advancements in the *instruction-level parallelism*, where the basic steps in instruction processing could be performed in a single cycle, most of the time. The RISC approach was straightforward to pipeline the stages of instruction processing so that an instruction is executed almost every cycle, on average. In order to utilize the parallelism in the hardware, it became necessary to issue multiple instructions in each cycle, called *superscalar* execution, as well as pipelining the processing of each instruction. Although superscalar execution will continue to be very important, barring some unforeseen breakthrough in instruction level parallelism, the leap to the *thread-level parallelism* is increasingly compelling as chips increase in capacity. Introducing thread-level parallelism on a single processor reduces the cost of the transition from one processor to multiple processors, thus making small scale SMPs more attractive on a broad scale. In additions, it establishes an architectural direction that may yield much grater latency tolerance in the long term. To conclude, there are strong indications that multithreading will be utilized in a future processor generations to hide the latency of local memory access [57].

REFERENCES

[1] F. ABOLHASSAN, R. DREFENSTEDT, J. KELLER, W.J. PAUL, AND D. SCHEERER, *On the physical design of PRAMs*, Computer Journal, 36 (1993), pp. 756–762.

[2] A. AGARWAL, *Performance tradeoffs in multithreaded processors*, IEEE Trans. Parallel and Distr. Syst., 3 (1992), pp. 525–539.

[3] A. AGARWAL, J. BABB, D. CHAIKEN, G. D'SOUZA, K. JOHNSON, D. KRANZ, J. KUBIATOWICZ, B.-H. LIM, G. MAA, AND K. MACKENZIE, *Sparcle: A multithreaded VLSI processor for parallel computing*, Lect. Notes Comput. Sc., 748, Springer-Verlag, Berlin, 1993, pp. 395.

[4] A. AGARWAL, R. BIANCHINI, D. CHAIKEN, K.L. JOHNSON, D. KRANZ, J. KUBIATOWICZ, B.-H. LIM, K. MACKENZIE, AND D. YEUNG, *The MIT Alewife machine: Architecture and performance*, in Proc. 22nd ISCA, June 1995, pp. 2–13.

[5] A. AGARWAL, G. D'SOUZA, K. JOHNSON, D. KRANZ, J. KUBIATOWICZ, K. KURIHARA, B.-H. LIM, G. MAA, D. NUSSBAUM, M. PARKIN, AND D. YEUNG, *The MIT Alewife machine: A large-scale distributed-memory multiprocessor*, in Proc. Workshop on Scalable Shared Memory Multiprocessors, 1991.

[6]   A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, *Sparcle: An evolutionary processor design for large-scale multiprocessors*, IEEE Micro, 13 (June 1993), pp. 48–61.

[7]   A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, *April: A processor architecture for multiprocessing*, in Proc. 17th ISCA, June 1990, pp. 104–114.

[8]   T. Agerwala and Arvind, *Data flow systems*, IEEE Computer, 15 (Feb. 1982), pp. 10–13.

[9]   G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith, *Exploiting heterogeneous parallelism on a multithreaded multiprocessor*, in Proc. 1992 Intl. Conf. Supercomputing, 1992, pp. 188–197.

[10] ———, *Integrated support for heterogeneous parallelism*, Multithreaded computer architecture: A summary of the state of the art, R.A. Iannucci, G.R. Gao, R. Halstead, and B. Smith, eds., Kluwer Academic, 1994.

[11] G. Alverson, S. Kahan, R. Korry, and C. McCann, *Scheduling on the Tera MTA*, Lect. Notes Comput. Sc., 949, Springer-Verlag, Berlin, 1995, pp. 19.

[12] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, *The Tera computer system*, in Proc. 1990 Intl. Conf. Supercomputing, June 1990, pp. 1–6.

[13] M. Amamiya, N. Takahashi, T. Naruse, and M. Yoshida, *A dataflow processor array system for solving partial differential equations*, in Proc. Intl. Symp. Applied Math. and Infor. Sci., March 1982.

[14] B.S. Ang, Arvind, and D. Chiou, *StarT the next generation: Integrating global caches and dataflow architecture*, Advanced Topics in Dataflow Computing and Multithreading, G.R. Gao, L. Bic, and J.-L. Gaudiot, eds., IEEE Computer Society Press, 1995, pp. 19–54.

[15] Arvind, L. Bic, and T. Ungerer, *Evolution of dataflow computers*, Advanced topics in data-flow computing, J.-L. Gaudiot and L. Bic, eds., Prentice Hall, 1991, pp. 3–33.

[16] Arvind and S.A. Brobst, *The evolution of dataflow architectures from static dataflow to P-RISC*, Intl. J. High Speed Computing, 5 (1993), pp. 125–153.

[17] Arvind and D.E. Culler, *Dataflow architectures*, Ann. Review in Comput. Sci., 1 (1986), pp. 225–253.

[18] ———, *Managing resources in a parallel machine*, Fifth Generation Computer Architectures, Elsevier Science Publishers, 1986, pp. 103–121.

[19] Arvind, K.P. Gostelow, and W. Plouffe, *An asynchronous programming language and computing machine*, Tech. Report 114a, Univ. California at Irvine, Dept. Information and Comput. Sci., Dec. 1978.

[20] Arvind and R.A. Iannucci, *A critique of multiprocessing von Neumann style*, in Proc. 10th ISCA, June 1983, pp. 426–436.

[21] Arvind and V. Kathail, *A multiple processor dataflow machine that supports generalized procedures*, in Proc. 8th ISCA, May 1981, pp. 291–302.

[22] Arvind, V. Kathail, and K. Pingali, *A processing element for a large multiprocessor dataflow machine*, in Proc. Intl. Conf. Circ. Comput., Oct. 1980.

[23] Arvind and R.S. Nikhil, *Executing a program on the MIT tagged-token dataflow architecture*, Lect. Notes Comput. Sc., 259, Springer-Verlag, Berlin, 1987, pp. 1–29.

[24] ———, *Executing a program on the MIT tagged-token dataflow architecture*, IEEE Trans. Computers, C-39 (1990), pp. 300–318.

[25] K. Asada, H. Terada, S. Matsumoto, S. Miyata, H. Asano, H. Miura, M. Shimizu, S. Komori, T. Fukuhara, and K. Shima, *Hardware structure of a one-chip data-driven processor: Q-p*, in Proc. 1987 ICPP, Aug. 1987, pp. 327–329.

[26] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grün, and C. Linchtenau, *Building the 4 processor SB-PRAM prototype*, in Proc. 30th Hawaii Intl. Conf. Syst. Sci., Jan. 1997, vol.5, pp. 14–23.

[27] J. Backus, *Can programming be librated form the von Neumann style? A functional style and its algebra of programs*, Comm. ACM, 21 (1978), pp. 613–641.

[28] P.M.C.C. Barahona and J.R. Gurd, *Simulated performance of the Manchester multi-ring dataflow machine*, in Proc. 2nd ICPC, Sep. 1985, pp. 419–424.

[29] ———, *Processor allocation in a multi-ring dataflow machine*, J. Parall. Distr. Comput., 3 (1986), pp. 67–85.

[30] S. Barkhordarian, *RAMPS: A realtime structured small-scale data flow system for parallel processing*, in Proc. 1987 ICPP, Aug. 1987, pp. 610–613.

[31] M. Beck, T. Ungerer, and E. Zehender, *Classification and performance evaluation of hybrid dataflow techniques with respect to matrix multiplication*, in Proc. GI/ITG Workshop PARS'93, April 1993, pp. 118–126.

[32] M.J. Beckerle, *An overview of the START (*T) computer system*, Tech. Report MCRC-TR-28, Motorola Technical Report, July 1992.

[33] _____, *Overview of the START (*T) multithreaded computer*, in Proc. COMPCON'93, 1993, pp. 148–156.

[34] L. BIC, *A process-oriented model for efficient execution of dataflow programs*, J. Parall. Distr. Comput., 8 (1990), pp. 42–51.

[35] L. BIC AND M. AL-MOUHAMED, *The EM-4 under implicit parallelism*, in Proc. 1993 Intl. Conf. Supercomputing, July 1993, pp. 21–26.

[36] A.P.W. BÖHM, J.R. GURD, AND Y.M. TEO, *The effect of iterative instructions in dataflow computers*, in Proc. 1989 ICPP, Aug. 1989, pp. 201–208.

[37] A.P.W. BÖHM AND J. SARGEANT, *Code optimization for tagged-token dataflow machine*, IEEE Trans. Computers, C-38 (1989), pp. 4–14.

[38] A.P.W. BÖHM AND Y.M. TEO, *Resource management in a multi-ring dataflow machine*, in Proc. CONPAR88, Sep. 1988, pp. B 191–200.

[39] R.F. BOOTHE, *Evaluation of multithreading and caching in large shared memory parallel computers*, Tech. Report UCB/CSD-93-766, Univ. California Berkeley, Comput. Sci. Division, July 1993.

[40] D.F. BRAILSFORD AND R.J. DUCKWORTH, *The MUSE machine – an architecture for structured data flow computation*, New Generation Computing, 3 (1985), pp. 181–195.

[41] R. BUEHRER AND K. EKANADHAM, *Incorporating data flow ideas into von Neumann processors for parallel processing*, IEEE Trans. Computers, C-36 (1987), pp. 1515–1522.

[42] M.L. CAMPBELL, *Static allocation for a dataflow multiprocessor*, in Proc. 1985 ICPP, Aug. 1985, pp. 511–517.

[43] W.W. CARLSON AND J.A.B. FORTES, *On the performance of combined data flow and control flow systems: Experiments using two iterative algorithms*, J. Parall. Distr. Comput., 5 (1988), pp. 359–382.

[44] A.J. CATTO AND J.R. GURD, *Resource management in dataflow*, in Proc. Conf. Functional Programming Lang. Comput. Arch., Oct. 1981, pp. 77–84.

[45] A.J. CATTO, J.R. GURD, AND C.C. KIRKHAM, *Non-deterministic dataflow programming*, in Proc. 6th ACM European Conf. Comput. Arch., April 1981, pp. 435–444.

[46] D. CHIOU, B.S. ANG, R. GREINER, ARVIND, J.C HOE, M.J. BECKERLE, J.E. HICKS, AND A. BOUGHTON, START-NG: *Delivering seamless parallel computing*, Lect. Notes Comput. Sc., 966, Springer-Verlag, Berlin, 1995, pp. 101–116.

[47] Y.M. CHONG, *Data flow chip optimizes image processing*, Comput. Design (Oct. 1984), pp. 97–103.

[48] R.P. COLWELL AND R.L. STEC, *A 0.6 μm BiCMOS processor with dynamic execution*, in Proc. Intl. Solid State Circuits Conf., Feb. 1995.

[49] D. COMTE AND N. HIFDI, *LAU multiprocessor: Microfunctional description and technologic choice*, in Proc. 1st Eur. Conf. Parallel and Distr. Proc., Feb. 1979, pp. 8–15.

[50] D. COMTE, N. HIFDI, AND J.C. SYRE, *The data driven LAU multiprocessor system: Results and perspectives*, in Proc. World Comput. Congress IFIP'80, Oct. 1980, pp. 175–180.

[51] M. CORNISH, *The TI dataflow architecture: The power of concurrency for avionics*, in Proc. 3rd Conf. Digital Avionics Syst., Nov. 1979, pp. 19–25.

[52] M. CORNISH, D.W. HOGAN, AND J.C. JENSEN, *The Texas Instruments distributed data processor*, in Proc. Lousiana Comput. Exposition, March 1979, pp. 189–193.

[53] D.E. CULLER, *Managing parallelism and resources in scientific dataflow programs*, Ph.D. thesis, MIT, Cambridge, MA, June 1989.

[54] D.E. CULLER, S. GOLDSTEIN, K.E. SCHAUSER, AND T. VON EICKEN, *TAM - A compiler controlled threaded abstract machine*, J. Parall. Distr. Comput., 18 (1993), pp. 347–370.

[55] D.E. CULLER AND G.M. PAPADOPOULOS, *The explicit token store*, J. Parall. Distr. Comput., 10 (1990), pp. 289–308.

[56] D.E. CULLER, A. SAH, K.E. SCHAUSER, T. VON EICKEN, AND J. WAWRZYNEK, *Fine-grain parallelism with minimal hardware support: A compiler-controlled Threaded Abstract Machine*, in Proc. 4th Intl. Conf. Arch. Support for Programming Lang. Operating Systems, April 1991, pp. 164–175.

[57] D.E. CULLER, J.P. SINGH, AND A. GUPTA, *Parallel Computer Architecture*, Morgan Kaufmann Publishers, 1998 (to appear).

[58] W.J. DALLY, J. FISKE, J. KEEN, R. LETHIN, M. NOAKES, P. NUTH, R. DAVISON, AND G. FYLER, *The message-driven processor: A multicomputer processing node with efficient mechanisms*, IEEE Micro, 12 (April 1992), pp. 23–39.

[59] J.D.G. DA SILVA AND J.V. WOODS, *Design of a processing subsystem forthe Manchester data flow computer*, in Proc. IEE, 128 (1981), pp. 218–224.

[60] _____, *A pseudoassociative matching storage using hardware hashing*, in Proc. IEE, 130 (1984), pp. 19–25.

[61] K. DAI AND W.K. GILOI, *A basic architecture supporting LGDF computation*, in Proc. 1990 Intl. Conf. Supercomputing, 1990, pp. 23–33.

[62] F. DAREMA, D.A. GEORGE, V.A. NORTON, AND G.F. PFISTER, *A single-program-multiple-data computational model for EPEX/FORTRAN*, Parallel Comput., 7 (1988), pp. 11–24.

[63] A.L. DAVIS, *The architecture and system method of DDM1: A recursively structured data driven machine*, in Proc. 5th ISCA, April 1978, pp. 210–215.

[64] A.L. DAVIS AND R.M. KELLER, *Data flow program graphs*, IEEE Computer, 15 (Feb. 1982), pp. 26–41.

[65] J.B. DENNIS, *First version of a data-flow procedure language*, Lect. Notes Comput. Sc., 19, Springer-Verlag, Berlin, 1974, pp. 362–376.

[66] ———, *The variaties of data flow computers*, in Proc. 1st Intl. Conf. Distr. Comput. Syst., Oct. 1979, pp. 430–439.

[67] ———, *Data flow supercomputers*, IEEE Computer, 13 (Nov. 1980), pp. 48–56.

[68] ———, *Dataflow computation: A case study*, Computer architecture - Concepts and systems (V.M. Milutinović, ed.), North-Holland, 1988, pp. 354–404.

[69] J.B. DENNIS, G.A. BOUGHTON, AND C.K.C. LEUNG, *Building blocks for data flow prototypes*, in Proc. 7th ISCA, May 1980, pp. 1–8.

[70] J.B. DENNIS AND G.R. GAO, *Multithreaded architectures: Principles, projects, and issues*, Multithreaded computer architecture: A summary of the state of the art, R.A. Iannucci, G.R. Gao, R. Halstead, and B. Smith, eds., Kluwer Academic, 1994, pp. 1–74.

[71] J.B. DENNIS, W.Y.P. LIM, AND W.B. ACKERMAN, *The MIT data flow engineering model*, in Proc. 9th World Comput. Congress IFIP'83, Sep. 1983, pp. 553–560.

[72] J.B. DENNIS AND D.P. MISUNAS, *A preliminary architecture for a basic data-flow processor*, in Proc. 2nd ISCA, Jan. 1975, pp. 126–132.

[73] K. DIEFFENDORF AND M. ALLEN, *Organization of the Motorola 88110 superscalar RISC microprocessor*, IEEE Micro, 12 (April 1992), pp. 40–63.

[74] T.A. DIEP, C. NELSON, AND J.P. SHEN, *Performance evaluation of the PowerPC 620 microprocessor*, in Proc. 22nd ISCA, June 1995, pp. 163–174.

[75] E.W. DIJKSTRA, *Co-operating sequential processes*, Programming Languages (F. Genuys, ed.), Academic Press, 1968, pp. 43–112.

[76] G.K. EGAN, *The CSIRAC II dataflow computer: Token and node definitions*, Tech. Report 31-009, School of Electrical Engineering, Swinburne Institute of Technology, 1990.

[77] G.K. EGAN, N.J. WEBB, AND A.P.W. BÖHM, *Some architectural features of the CSIRAC II data-flow computer*, Advanced topics in data-flow computing, J.-L. Gaudiot and L. Bic, eds., Prentice Hall, 1991, pp. 3–33.

[78] K.G. EGAN AND S. WATHANASIN, *Data-driven computation: Its suitability in control systems*, in Proc. Science Research Council's Workshop on Distr. Comput., July 1978.

[79] S.J. EGGERS, J.S. EMER, H.M. LEVY, J.L. LO, R.L. STAMM, AND D.M. TULLSEN, *Simultaneous multithreading: A foundation for next-generation processors*, IEEE Micro, 17 (Oct. 1997).

[80] P. EVRIPIDOU AND J.-L. GAUDIOT, *The USC decoupled multilevel data-flow execution model*, Advanced topics in data-flow computing, J.-L. Gaudiot and L. Bic, eds., Prentice Hall, 1991, pp. 347–379.

[81] M.R. EXUM AND J.-L. GAUDIOT, *Network design and allocation consideration in the Hughes data-flow machine*, Parallel Comput., 13 (1990), pp. 17–34.

[82] M. FLYNN, *Some computer organizations and their effectiveness*, IEEE Trans. Computers, C-21 (1972), pp. 948–960.

[83] A. FORMELLA, J. KELLER, AND T. WALLE, *HPP: A high performance PRAM*, Lect. Notes Comp. Sc., 1123, Springer-Verlag, Berlin, 1996, pp. 425–434.

[84] ———, *HPP: A high performance PRAM*, Tech. Report H 124, 02/96, University of Saarbruecken, 1996.

[85] T. FUJITA, *A multithreaded processor architecture for parallel symbolic computation*, Tech. Report MIT/LCS/TM-338, CS Lab., MIT, Cambridge, MA, 1987.

[86] G.R. GAO, *An efficient hybrid dataflow architecture model*, J. Parall. Distr. Comput., 19 (1993), pp. 293–306.

[87] G.R. GAO, J.-L. GAUDIOT, AND L. BIC, *Dataflow and multithreaded architectures: Guest editor's introduction*, J. Parall. Distr. Comput., 18 (1993), pp. 271–272.

[88] J.-L. GAUDIOT AND L. BIC, *Advanced topics in dataflow computing*, Prentice Hall, 1991.

[89] J.-L. GAUDIOT, R.W. VEDDER, G.K. TUCKER, D. FINN, AND M.L. CAMPBELL, *A distributed VLSI architecture for efficient signal and data processing*, IEEE Trans. Computers, C-34 (1985), pp. 1072–1087.

[90] D. GHOSAL AND L.N. BHUYAN, *Performance analysis of the MIT tagged token dataflow ar-

chitecture, in Proc. 1987 ICPP, Aug. 1987, pp. 680–683.

[91] W.K. GILOI, U. BRÜNING, AND W. SCHRÖDER-PREIKSCHAT, *MANNA: Prototype of a distributed memory architecture with maximized sustained performance*, in Proc. 5th Euromicro Workshop on Prallel and Distributed Processing, Jan. 1996.

[92] J.R.W. GLAUERT, J.R. GURD, AND C.C. KIRKHAM, *Evolution of dataflow architecture*, in Proc. IFIP WG 10.3 Workshop on Hardware Supported Implementation on Concurrent Lang. in Distr. Systems, March 1984, pp. 1–18.

[93] J.R.W. GLAUERT, J.R. GURD, C.C. KIRKHAM, AND I. WATSON, *The dataflow approach*, Distributed Computing, F.B. Chambers, D.A. Duce, and G.P. Jones, eds., Academic Press, 1984, pp. 1–53.

[94] R.R. GLENN AND D.V. PRYOR, *Instrumentation for a massively parallel MIMD application*, in Proc. Intl. Conf. Measurement and Modeling of Comput. Syst., May 1991, pp. 208–209.

[95] E. GLÜCK-HILTROP, *The Stollman dataflow machine*, in Proc. CONPAR88, Sep. 1988.

[96] E. GLÜCK-HILTROP, M. RAMLOW, AND U. SCHÜRFELD, *The Stollman dataflow machine*, Lect. Notes Comput. Sc., 365, Springer-Verlag, Berlin, 1989, pp. 433–457.

[97] K.P. GOSTELOW AND ARVIND, *The U-interpreter*, IEEE Computer, 15 (Feb. 1982), pp. 42–49.

[98] K.P. GOSTELOW AND R.E. THOMAS, *A view of dataflow*, in Proc. National Comput. Conf., June 1979, pp. 629–636.

[99] ———, *Performance of a simulated dataflow computer*, IEEE Trans. Computers, C-29 (1980), pp. 905–919.

[100] V.G. GRAFE AND J.E. HOCH, *The Epsilon-2 multiprocessor system*, J. Parall. Distr. Comput., 10 (1990), pp. 309–318.

[101] V.G. GRAFE, J.E. HOCH, G.S. DAVIDSON, V.P. HOLMES, D.M. DAVENPORT, AND K.M. STEELE, *The Epsilon project*, Advanced topics in data-flow computing, J.-L. Gaudiot and L. Bic, eds., Prentice Hall, 1991, pp. 175–205.

[102] J.D. GRIMM, J.A. EGGERT, AND G.W. KARCHER, *Distributed signal processing using data flow techniques*, in Proc. 17th Hawaii Intl. Conf. Syst. Sci., Jan. 1984, pp. 29–38.

[103] W. GRUENEWALD AND T. UNGERER, *Towards extremely fast context switching in a block-multithreaded processor*, in Proc. 22nd Euromicro Conf., Sep. 1996, pp. 592–599.

[104] ———, *A multithreaded processor designed for distributed shared memory systems*, in Proc. Intl. Conf. Advances in Parallel and Distr. Comput., March 1997.

[105] M. GULATI AND N. BAGHERZADEH, *Performance study of a multithreaded superscalar microprocessor*, in Proc. 2nd Intl. Symp. High-Performance Comput. Arch., Feb. 1996, pp. 291–301.

[106] A. GUPTA, J. HENNESSY, K. GHARACHORLOO, T. MOWRY, AND W.-D. WEBER, *Comparative evaluation of latency reducing and tolerating techniques*, in Proc. 18th ISCA, May 1991, pp. 254–263.

[107] J.R. GURD, *The Manchester dataflow machine*, Future Generations Computer Systems, 1 (1985), pp. 201–212.

[108] J.R. GURD, C.C. KIRKHAM, AND I. WATSON, *The Manchester prototype dataflow computer*, Comm. ACM, 28 (1985), no. 1, pp. 34–52.

[109] J.R. GURD AND I. WATSON, *A multilayered data flow computer architecture*, in Proc. 1977 ICPP, Aug. 1977, pp. 94.

[110] ———, *Data driven system for high speed computing, Part I: Structuring software for parallel execution*, Computer Design, 19 (1980), pp. 91–96.

[111] ———, *Data driven system for high speed computing, Part II: Hardware design*, Computer Design, 19 (1980), pp. 97–106.

[112] ———, *Preliminary evaluation of a prototype dataflow computer*, in Proc. 9th World Comput. Congress IFIP'83, Sep. 1983, pp. 545–551.

[113] L. GWENNAP, *DanSoft develops WVLIW design*, Microprocessor Report, 11 (Feb. 17, 1997), pp. 18-22.

[114] R.H. HALSTEAD, JR., *Implementation of Multilisp: Lisp on a multiprocessor*, in Proc. ACM Symp. Lisp and Functional Programming, Aug. 1984, pp. 9–17.

[115] ———, *Multilisp: A language for concurrent symbolic computation*, ACM Trans. Programming Lang. Syst., 7 (1985), pp. 501–538.

[116] ———, *Parallel computing using Multilisp*, Parallel computation and computers for artificial intelligence (J. Kowalik, ed.), Kleuwer Academic, 1988, pp. 21–49.

[117] R.H. HALSTEAD, JR. AND T. FUJITA, *MASA: A multithreaded processor architecture for parallel symbolic computing*, in Proc. 15th ISCA, May 1988, pp. 443–451.

[118] R. HEMPEL, A.J.G. HEY, O. MCBRYAN, AND D.W. WALKER, *Message passing interfaces (special issue)*, Parallel Comput., 20 (1994), pp. 417–673.

[119] J. L. HENNESSY AND D. A. PATTERSON, *Computer architecture a quantitative approach*, Mor-

gan Kaufmann, 1996.

[120] K. HIRAKI, K. NISHIDA, S. SEKIGUCHI, AND T. SHIMADA, *Maintenance architecture and LSI implementation of a dataflow computer with a large number of processors*, in Proc. 1986 ICPP, Aug. 1986, pp. 584–591.

[121] K. HIRAKI, S. SEKIGUCHI, AND T. SHIMADA, *Status report of SIGMA-1: A dataflow supercomputer*, Advanced topics in data-flow computing, J.-L. Gaudiot and L. Bic, eds., Prentice Hall, 1991, pp. 207–223.

[122] K. HIRAKI, T. SHIMADA, AND K. NISHIDA, *A hardware design of the SIGMA-1, a data flow computer for scientific computations*, in Proc. 1984 ICPP, Aug. 1984, pp. 524–531.

[123] H. HIRATA, K. KIMURA, S. NAGAMINE, Y. MOCHIZUKI, A. NISHIMURA, Y. NAKASE, AND T. NISHIZAWA, *An elementary processor architecture with simultaneous instruction issuing from multiple threads*, in Proc. 19th ISCA, May 1992, pp. 136–145.

[124] H. HIRATA, Y. MOCHIZUKI, A. NISHIMURA, Y. NAKASE, AND T. NISHIZAWA, *A multithreaded processor architecture with simultaneous instruction issuing*, Supercomputer, 49 (1992), pp. 23–39.

[125] J.L.A. HUGHES, *Implementing control-flow structures in dataflow programs*, in Proc. COMPCON Spring 82, Feb. 1982, pp. 87–90.

[126] H.H.J. HUM, K.B. THEOBALD, AND G.R. GAO, *Building multithreaded architectures with off-the-shelf microprocessor*, in Proc. IPPS 94, April 1994, pp. 288–294.

[127] R.A. IANNUCCI, *Toward a dataflow/von Neumann hybrid architecture*, in Proc. 15th ISCA, May 1988, pp. 131–140.

[128] R.A. IANNUCCI, G.R. GAO, R. HALSTEAD, AND B. SMITH, *Multithreaded computer architecture: A summary of the state of the art*, Kluwer Academic, 1994.

[129] Y. INAGAMI AND J.F. FOLEY, *The specification of a new Manchester dataflow machine*, in Proc. 1989 Intl. Conf. Supercomputing, June 1989, pp. 371–380.

[130] N. ITO, M. KISHI, E. KUNO, AND K. ROKUSAWA, *The data-flow based parallel inference machine to support two basic languages in KL1*, in Proc. IFIP TC-10 Working Conf. Fifth Generation Comput. Arch., July 1985, pp. 123–145.

[131] N. ITO AND K. MUSADA, *Parallel inference machine based on the data flow model*, in Proc. Intl. Workshop High Level Comput. Arch., May 1984.

[132] N. ITO, M. SATO, E. KUNO, AND K. ROKUSAWA, *The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D*, in Proc. 13th ISCA, June 1986, pp. 149–156.

[133] M. IWASHITA AND T. TEMMA, *Experiments on a data flow machine*, in Proc. IFIP Conf., 1987, pp. 235–245.

[134] T. JEFFERY, *The μPD7281 processor*, Byte (Nov. 1985), pp. 237–246.

[135] I. JEREBIC, B. SLIVNIK, AND R. TROBEC, *Library for programming on tourus-connected multiprocessors*, in Proc. PACTA'92, Sep. 1992, pp. 386–395.

[136] C. JESSHOPE, *Scalable parallel computers*, EURO-PAR Conf. (Stockholm, Sweden), Aug. 1995, Tutorial.

[137] H.F. JORDAN, *Performance measurements on HEP: A pipelined MIMD computer*, in Proc. 10th ISCA, June 1983, pp. 207–212.

[138] K.M. KAVI, B.P. BUCKLES, AND U.N. BHAT, *A formal definition of data flow graph model*, IEEE Trans. Computers, C-35 (1986), pp. 940–948.

[139] K. KAWAKAMI AND J.R. GURD, *A scalable dataflow structure store*, in Proc. 13th ISCA, June 1986, pp. 243–250.

[140] M. KISHI, H. YASUHARA, AND Y. KAWAMURA, *DDDP: A distributed data driven processor*, in Proc. 10th ISCA, June 1983, pp. 236–242.

[141] D. KLAPPHOLZ, Y. LIAO, D. J. WANG, AND A. OMONDI, *Toward a hybrid data-flow/control-flow MIMD architecture*, in Proc. 5th Intl. Conf. Distr. Comput. Syst., May 1985, pp. 10–15.

[142] Y. KODAMA, Y. KOUMURA, M. SATO, H. SAKANE, S. SAKAI, AND Y. YAMAGUCHI, *EMC-Y: Parallel processing element optimizing communication and computation*, in Proc. 1993 Intl. Conf. Supercomputing, July 1993, pp. 167–174.

[143] Y. KODAMA, H. SAKANE, M. SATO, H. YAMANA, S. SAKAI, AND Y. YAMAGUCHI, *The EMC-X parallel computer: Architecture and basic performance*, in Proc. 22nd ISCA, June 1995, pp. 14–23.

[144] S. KOMORI, K. SHIMA, S. MIYATA, AND H. TERADA, *Parallel processing with large grain data flow techniques*, IEEE Micro, 9 (June 1989), pp. 45–59.

[145] J.S. KOWALIK, *Parallel MIMD computation: The HEP supercomputer and its applications*, MIT Press, 1985.

[146] D.A. KRANZ, R.H. HALSTEAD, AND E. MOHR, *Mul-T: A high-performance parallel Lisp*, ACM

SIGPLAN Notices, 24 (1989), no. 7, pp. 81–90.

[147] J. KUBIATOWICZ AND A. AGARWAL, *Anatomy of a message in the Alewife multiprocessor*, in Proc. 1993 Intl. Conf. Supercomputing, July 1993, pp. 195–206.

[148] J.T. KUEHN AND B.J. SMITH, *The Horizon supercomputing system: architecture and software*, in Proc. Supercomputing '88, Nov. 1988, pp. 28–34.

[149] A. KUMAR, *The HP PA-8000 RISC CPU*, IEEE Micro, 17 (March/April 1997), pp. 27–32.

[150] J. LAUDON AND D. LENOSKI, *The SGI Origin: A ccNUMA highly scalable server*, in Proc. 24th ISCA, June 1997, pp. 241-251.

[151] B. LEE AND A.R. HURSON, *Dataflow architectures and multithreading*, IEEE Computer, 27 (Aug. 1994), pp. 27–39.

[152] J.L. LO, S.J. EGGERS, J.S. EMER, H.M. LEVY, R.L. STAMM, AND D.M. TULLSEN, *Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading*, ACM Trans. Comput. Systems, 15 (1997).

[153] M. LOIKKANEN AND N. BAGHERZADEH, *A fine-grain multithreading superscalar architecture*, in Proc. PACT'96, Oct. 1996, pp. 163–168.

[154] T.E. MANKOVIC, V. POPESCU, AND H. SULLIVAN, *CHoPP priciples of operations*, in Proc. 2nd Intl. Supercomputer Conf., May 1987, pp. 2–10.

[155] O.C. MAQUELIN, H.H.J. HUM, AND G.R. GAO, *Costs and benefits of multithreading with off-the-shelf processors*, Lect. Notes Comp. Sc., 966, Springer-Verlag, Berlin, 1995, pp. 117–128.

[156] A. MIKSCHL AND W. DAMM, *Msparc: A multithreaded Sparc*, Lect. Notes Comp. Sc., 1123, Springer-Verlag, Berlin, 1996, pp. 461–469.

[157] J. MILEWSKI, *Data driven control in I-structures*, in Proc. 2nd Intl. Conf. Parallel Computing, Sep. 1985, pp. 377–382.

[158] J. MILEWSKI AND I. MORI, *Efficient loop handling in a stream-oriented unraveling dataflow interpreter*, Comput. Artif. Intell., 9 (1990), pp. 545–570.

[159] S.W. MOORE, *Multithreaded processor design*, Kluwer Academic Publishers Boston, 1996.

[160] V.L. NARASIMHAN AND T. DOWNS, *Operating system features of a dynamic dataflow array processing system (PATTSY)*, in Proc. 3rd Annual Parallel Processing Symp., March 1989, pp. 722–740.

[161] S.S. NEMAWARKAR AND G.R. GAO, *Measurement and modeling of EARTH-MANNA multi-threaded architecture*, in Proc. 4th Intl. Workshop MASCOTS'96, Feb. 1996, pp. 109–114.

[162] R.S. NIKHIL, *The parallel programming language Id and its compilation for parallel machines*, Intl. J. High Speed Computing, 5 (1993), pp. 171–223.

[163] ———, *A multithreaded implementation of Id using P-RISC graphs*, Lect. Notes Comput. Sc., 768, Springer-Verlag, Berlin, 1994, pp. 390–405.

[164] R.S. NIKHIL AND ARVIND, *Can dataflow subsume von Neumann computing?*, in Proc. 16th ISCA, May 1989, pp. 262–272.

[165] R.S. NIKHIL, G.M. PAPADOPOULOS, AND ARVIND, **T: A multithreaded massively parallel architecture*, in Proc. 19th ISCA, May 1992, pp. 156–167.

[166] H. NISHIKAWA, H. TERADA, K. KOMATSU, S. YOSHIDA, T. OKAMOTO, Y. TSUJI, S. TAKAMURA, T. TOKURA, Y. NISHIKAWA, S. HARA, AND M. MEICHI, *Architecture of a one-chip data-driven processor: Q-p*, in Proc. 1987 ICPP, Aug. 1987, pp. 319–326.

[167] M.D. NOAKES, D.A. WALLACH, AND W.J. DALLY, *The J-Machine multicomputer: An architectural evaluation*, in Proc. 20th ISCA, May 1993, pp. 224–236.

[168] M. OJSTERŠEK, V. ŽUMER, AND P. KOKOL, *Data flow computer models*, in Proc. CompEuro '87, May 1987, pp. 884–885.

[169] A. OMONDI AND D. KLAPPHOLZ, *Combining control driven and data driven computation for high processing rate*, in Proc. Intl. Computing Symp., March 1985.

[170] S. PALACHARLA, N.P. JOUPPI, AND J.E. SMITH, *Complexity-effective superscalar processors*, in Proc. 24th ISCA, June 1997, pp. 206-218.

[171] G.M. PAPADOPOULOS, *Implementation of a general-purpose dataflow multiprocessor*, Tech. Report TR-432, MIT Laboratory for Computer Science, Cambridge, Ma., Aug. 1988.

[172] G.M. PAPADOPOULOS AND D.E. CULLER, *Monsoon: An explicit token-store architecture*, in Proc. 17th ISCA, June 1990, pp. 82–91.

[173] G.M. PAPADOPOULOS AND K.R. TRAUB, *Multithreading: A revisionist view of dataflow architectures*, in Proc. 18th ISCA, May 1991, pp. 342–351.

[174] L.M. PATNAIK, R. GOVINDARAJAN, AND N.S. RAMADOSS, *Design and performance evaluation of EXMAN: EXtended MANchester data flow computer*, IEEE Trans. Computers, C-35 (1986), pp. 229–244.

[175] A. PLAS, D. COMTE, O. GELLY, AND J. C. SYRE, *LAU system architecture: A parallel data driven processor based on single assignment*, in Proc. 1976 ICPP, Aug. 1976, pp. 293–302.

[176] G.M. QUÉNOT AND B. ZAVIDOVIQUE, *A data-flow processor for real-time low-level image processing*, in Proc. IEEE Custom Integrated Circuits Conf., May 1991, pp. 1241–1244.

[177] J.E. REQUA, *The piecewise data flow architecture control flow and register management*, in Proc. 10th ISCA, June 1983, pp. 84–89.

[178] J.E. REQUA AND J.R. MCGRAW, *The piecewise data flow architecture: Architectural concept*, IEEE Trans. Computers, C-32 (1983), pp. 425–438.

[179] J. RISAU, A. MIKSCHL, AND W. DAMM, *A RISC approach to weak cache coherence*, Lect. Notes Comp. Sc., 1123, Springer-Verlag, Berlin, 1996, pp. 453–456.

[180] L. ROH AND W.A. NAJJAR, *Design of a storage hierarchy in multithreaded architectures*, in Proc. 28th MICRO, 1995, pp. 271–278.

[181] S. SAKAI, *Synchronization and pipeline design for a multithreaded massively parallel computer*, Advanced Topics in Dataflow Computing and Multithreading, G.R. Gao, L. Bic, and J.-L. Gaudiot, eds., IEEE Computer Society Press, 1995, pp. 55–74.

[182] S. SAKAI, K. OKAMOTO, H. MATSUOKA, H. HIRONO, Y. KODAMA, AND M. SATO, *Super-threading: Architectural and software mechanisms for optimizing parallel computation*, in Proc. 1993 Intl. Conf. Supercomputing, July 1993, pp. 251–260.

[183] S. SAKAI, Y. YAMAGUCHI, K. HIRAKI, Y. KODAMA, AND T. YUBA, *An architecture of a dataflow single chip processor*, in Proc. 16th ISCA, May 1989, pp. 46–53.

[184] J. SARGEANT AND C.C. KIRKHAM, *Stored data structures on the Manchester dataflow machines*, in Proc. 13th ISCA, June 1986, pp. 235–242.

[185] A.V.S. SASTRY, L.M. PATNAIK, AND J. ŠILC, *Dataflow architectures for logic programming*, Electrotehnical Review, 55 (1988), pp. 9–19.

[186] M. SATO, Y. KODAMA, S. SAKAI, AND Y. YAMAGUCHI, *EM-C: Programming with explicit parallelism and locality for the EM-4 multiprocessor*, in Proc. PACT'94, Aug. 1994, pp. 3–14.

[187] ――――, *Experiences with executing shared memory programs using fine-grained communication and multithreading in EM-4*, in Proc. IPPS 94, April 1994, pp. 630–636.

[188] M. SATO, Y. KODAMA, S. SAKAI, Y. YAMAGUCHI, AND Y. KOUMURA, *Thread-based programming for the EM-4 hybrid dataflow machine*, in Proc. 19th ISCA, May 1992, pp. 146–155.

[189] M. SATO, Y. KODAMA, S. SAKAI, Y. YAMAGUCHI, AND S. SEKIGUCHI, *Distributed data structure in thread-based programming for a highly parallel dataflow machine EM-4*, Advanced Topics in Dataflow Computing and Multithreading, G.R. Gao, L. Bic, and J.-L. Gaudiot, eds., IEEE Computer Society Press, 1995, pp. 131–142.

[190] J. SÉROT, G.M. QUÉNOT, AND B. ZAVIDOVIQUE, *A functional data-flow architecture dedicated to real-time image processing*, IFIP Trans., A-23 (1993), pp. 129–140.

[191] J.A. SHARP, *Data flow computing*, Ellis Horwood Ltd. Publishers, 1985.

[192] A. SHAW, ARVIND, AND R.P. JOHNSON, *Performance tuning scientific codes for dataflow execution*, in Proc. PACT'96, Oct. 1996, pp. 198–207.

[193] T. SHIMADA, K. HIRAKI, K. NISHIDA, AND S. SEKIGUCHI, *Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations*, in Proc. 13th ISCA, June 1986, pp. 226–234.

[194] U. SIGMUND AND T. UNGERER, *Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip*, in Proc. 4th Parallel Syst. Algorithms Workshop, April 1996.

[195] ――――, *Identifying bottlenecks in multithreaded superscalar multiprocessor*, Lect. Notes Comp. Sc., 1123, Springer-Verlag, Berlin, 1996, pp. 797–800.

[196] J. ŠILC AND B. ROBIČ, *The review of some data flow computer architectures*, Informatica, 11 (1987), pp. 61–66.

[197] ――――, *Efficient dataflow architecture for specialized computations*, in Proc. 12th World Congress on Scientifics Computation, July 1988, pp. 4.681–4.684.

[198] ――――, *Synchronous dataflow-based architecture*, Microproc. Microprog., 27 (1989), pp. 315–322.

[199] ――――, *MADAME – Macro-dataflow machine*, in Proc. MELECON '91, May 1991, pp. 985–988.

[200] J. ŠILC, B. ROBIČ, AND L.M. PATNAIK, *Performance evaluation of an extended static dataflow architecture*, Comput. Artif. Intell., 9 (1990), pp. 43–60.

[201] B.J. SMITH, *A pipelined, shared resource MIMD computer*, in Proc. 1978 ICPP, Aug. 1978, pp. 6–8.

[202] ――――, *Architecture and applications of the HEP multiprocessor computer system*, SPIE Real-Time Signal Processing IV, 298 (1981), pp. 241–248.

[203] ――――, *The architecture of HEP*, Parallel MIMD computation: HEP supercomputer and its application (J.S. Kowalik, ed.), MIT Press, 1985, pp. 41–55.

[204] D.F SNELLING, *The design and analysis of a Stateless Data-Flow Architectures*, Tech. Report

UMCS-93-7-2, University of Manchester, Department of Computer Science, 1993.

[205] D.F SNELLING AND G.K. EGAN, *A comparative study of data-flow architectures*, Tech. Report UMCS-94-4-3, University of Manchester, Department of Computer Science, 1994.

[206] A. SOHN, C. KIM, AND M. SATO, *Multithreading with the EM-4 distributed-memory multiprocessor*, in Proc. PACT'95, June 1995, pp. 27–36.

[207] S. P. SONG, M. DENMAN, AND J. CHANG, *The PowerPC 604 RISC microprocessor*, IEEE Micro, 14 (Oct. 1994), pp. 8–17.

[208] V.P. SRINI, *An architectural comparison of dataflow systems*, IEEE Computer, 19 (March 1986), pp. 68–88.

[209] J. STROHSCHNEIDER, B. KLAUER, AND K. WALDSCHMIDT, *An associative communication network for fine and large grain dataflow*, in Proc. Euromicro Workshop on Parallel and Distr. Processing, 1995, pp. 324–331.

[210] J. STROHSCHNEIDER, B. KLAUER, S. ZICKENHEIMER, AND K. WALDSCHMIDT, *Adarc: A fine grain dataflow architecture with associative communication network*, in Proc. 20th Euromicro Conf., Sep. 1994, pp. 445-450.

[211] J.C. SYRE, D. COMTE, AND N. HIFDI, *Pipelining, parallelism and asynchronism in the LAU system*, in Proc. 1977 ICPP, Aug. 1977, pp. 87–92.

[212] N. TAKAHASHI AND M. AMAMIYA, *A data flow processor array system: Design and analysis*, in Proc. 10th ISCA, June 1983, pp. 243–250.

[213] T. TEMMA, *Dataflow processor for image processing*, Intl. Journal of Mini and Microcomputers **5** (1980), no. 3, 52–56.

[214] T. TEMMA, M. IWASHITA, K. MATSUMOTO, H. KUROKAWA, AND T. NUKIYAMA, *Data flow processor chip for image processing*, IEEE Trans. Elec. Dev., ED-32 (1985), pp. 1784–1791.

[215] M. THISTLE AND B.J. SMITH, *A processor architecture for Horizon*, in Proc. Supercomputing '88, Nov. 1988, pp. 35–41.

[216] S.A. THORESON, A.N. LONG, AND J.R. KERNS, *Performance of three dataflow computers*, in Proc. 14th Ann. Comput. Sci. Conf., Feb. 1986, pp. 93–99.

[217] K.R. TRAUB, G.M. PAPADOPOULOS, M.J. BECKERLE, J.E. HICKS, AND J. YOUNG, *Overwiev of the Monsoon project*, in Proc. 1991 Intl. Conf. Comput. Design, 1991, pp. 150–155.

[218] P.C. TRELEAVEN, *Principal components of a data flow computer*, in Proc. 1978 Euromicro Symp., Oct. 1978, pp. 366–374.

[219] P.C. TRELEAVEN, D.R. BROWNBRIDGE, AND R.P. HOPKINS, *Data-driven and demand-driven computer architectures*, Computing Surveys, 14 (1982), pp. 93–143.

[220] P.C. TRELEAVEN, R.P. HOPKINS, AND P.W. RAUTENBACH, *Combining data flow and control flow computing*, Computer Journal, 25 (1982), pp. 207–217.

[221] D.M. TULLSEN, S.J. EGGERS, J.S. EMER, H.M. LEVY, J.L. LO, AND R.L. STAMM, *Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor*, in Proc. 23th ISCA, May 1996, pp. 191–202.

[222] D.M. TULLSEN, S.J. EGGERS, AND H.M. LEVY, *Simultaneous multithreading: Maximizing on-chip parallelism*, in Proc. 22nd ISCA, June 1995, pp. 392–403.

[223] T. UNGERER AND E. ZEHENDNER, *A multi-level parallelism architecture*, Tech. Report 230, Univ. Augsburg, Institute for Mathematics, 1991.

[224] ⸻, *Threads and subinstruction level parallelism in a data flow architecture*, Lect. Notes Comput. Sc., 634, Springer-Verlag, Berlin, 1992, pp. 731–736.

[225] R. VEDDER, M. CAMPBELL, AND G. TUCKER, *The Hughes data flow multiprocessor*, in Proc. 5th Intl. Conf. Distr. Comput. Syst., May 1985, pp. 2–9.

[226] R. VEDDER AND D. FINN, *The Hughes data flow multiprocessor: Architecture for efficient signal and data processing*, in Proc. 12th ISCA, June 1985, pp. 324–332.

[227] A.H. VEEN AND R. VAN DEN BORN, *The RC compiler for the DTN dataflow computer*, J. Parall. Distr. Comput., 10 (1990), pp. 319–322.

[228] D.W. WALL, *Limits of instruction-level parallelism*, in Proc. 4th Intl. Conf. Arch. Support for Programming Languages and Operating Syst., April 1991, pp. 176–188.

[229] I. WATSON AND J.R. GURD, *A prototype data flow computer with token labelling*, in Proc. National Comput. Conf., June 1979, pp. 623–628.

[230] I. WATSON AND J.R. GURD, *A practical data flow computer*, IEEE Computer, 15 (Feb. 1982), pp. 51–57.

[231] K.C. YEAGER, *The MIPS R10000 superscalar microprocessor*, IEEE Micro, 16 (April 1996), pp. 28–40.

[232] T. YUBA, T. SHIMADA, K. HIRAKI, AND H. KISHIWAGI, *SIGMA-1: A dataflow computer for scientific computations*, Comput. Physics Comm., 37 (1985), pp. 141–148.

[233] A.C. YUCETURK, B. KLAUER, S. ZICKENHEIMER, R. MOORE, AND K. WALDSCHMIDT, *Mapping*

of neural networks onto dataflow graphs, in Proc. 22nd Euromicro Conf., Sep. 1996, pp. 51–57.

[234] E. Zehendner and T. Ungerer, *The ASTOR architecture*, in Proc. 7th Intl. Conf. Distr. Comput. Syst., Sep. 1987, pp. 424–430.

[235] ———, *A large-grain data flow architecture utilizing multiple levels of parallelism*, in Proc. CompEuro '92, May 1992, pp. 23–28.