Beyond Dataflow

Borut Robič[1], Jurij Šilc[2] and Theo Ungerer[3]

[1] Faculty of Computer and Information Science, University of Ljubljana, Slovenia
[2] Computer Systems Department, Jožef Stefan Institute, Ljubljana, Slovenia
[3] Department of Computer Design and Fault Tolerance, University of Karlsruhe, Germany

This paper presents some recent advanced dataflow architectures. While the dataflow concept offers the potential of high performance, the performance of an actual dataflow implementation can be restricted by a limited number of functional units, limited memory bandwidth, and the need to associatively match pending operations with available functional units. Since the early 1970s, there have been significant developments in both fundamental research and practical realizations of dataflow models of computation. In particular, there has been active research and development in multithreaded architectures that evolved from the dataflow model. Also some other techniques for combining control-flow and dataflow emerged, such as coarse-grain dataflow, dataflow with complex machine operations, RISC dataflow, and micro dataflow. These developments have also had certain impact on the conception of high-performance superscalar processors in the "post-RISC" era.

Keywords: Coarse-grain dataflow, computer architecture, hybrid von Neumann/dataflow, micro dataflow, RISC dataflow, superscalar microprocessor, survey, threaded dataflow.

1. Introduction

The most common computing model (i.e., a description of how a program is to be evaluated) is the von Neumann control-flow computing model. This model assumes that a program is a series of addressable instructions, each of which either specifies an operation along with memory locations of the operands or it specifies (un)conditional transfer of control to some other instruction. A control-flow computing model essentially specifies the next instruction to be executed depending on what happened during the execution of the current instruction. The next instruction to be executed is pointed to and triggered by the program counter. This instruction is executed even if some of its operands are not available yet (e.g., uninitialized).

The dataflow model represents a radical alternative to the von Neumann computing model since the execution is driven only by the availability of operands. It has no program counter and global updatable store, i.e., the two features of the von Neumann model that become bottlenecks in exploiting parallelism. The serialization of the von Neumann computing model is a serious limitation for exploiting more parallelism in today's microprocessors – e.g., superscalars. In dataflow computing parallelism is limited only by the actual data dependences between instructions in the application program. Since program execution is driven only by the availability of the operands (at the inputs to the functional units), dataflow computers have the potential for exploiting all the parallelism available in a program. Namely, the firing rule of the dataflow model, which specifies when an instruction can actually be executed, states that an instruction is enabled as soon as corresponding operands are present and executed when hardware resources are available. Because there is no need for a program counter, dataflow architectures represent a radical alternative to the von Neumann architecture. Dataflow computers use dataflow graphs as their machine language. Dataflow graphs, as opposed to conventional machine languages, specify only a partial order for the instruction execution and thus provide opportunities for parallel and pipelined execution at the level of individual instructions.

Fine-grain dataflow computers, which are based on the single-token-per-arc (static) approach,

	von Neumann	Dataflow	Hybrid
instruction scheduling	total sequencing	based on data dependences	partial ordering can be explicitly expressed
hiding long latency	cannot hide	by split-phase memory operation	by context switching
synchronization support	unefficient At low level	too general intra-procedure communication	dataflow used only at inter-procedural level

Table 1. Von Neumman, dataflow and hybrid approach.

tagged-token (dynamic) approach or explicit token store approach have circular pipelined organization, and usually perform quite poorly with sequential code (Arvind et al., 1991). This is because an instruction of the same execution thread can only be issued to the dataflow pipeline after the completion of its predecessor instruction. In case of an 8-stage dataflow pipeline, for example, instructions of the same thread can be issued at the most every eight cycles. If the computation load is low, for instance for a single sequential thread, the utilization of the dataflow processor drops to one eighth of its maximum performance.

Another disadvantage of the fine-grain dataflow is the overhead associated with token matching. For example, before a dyadic instruction is issued to the execution stage, two operands (each packed in the so-called token) must be present. The first arrived token is stored in the waiting-matching store and only when the second token arrives the instruction can be issued (i.e., fired). This introduces a *bubble* in the execution stage(s) of the dataflow processor pipeline, i.e., a sequence of idle pipeline stages. Clearly, this may affect the system's performance, so bubbles should not be neglected. For example, the pipeline bubbles summed up to 28.75 % of the total execution time when solving the Traveling Salesman problem on the Monsoon dataflow machine (Papadopoulos and Culler, 1990). Since a context switch occurs in such fine-grain dataflow after each instruction execution, no use of registers is possible for optimizing the access time to data in order to avoid pipeline bubbles caused by dyadic instructions, and for reducing the total number of tokens during program execution.

A solution to these problems is to combine dataflow with control-flow mechanisms. Actually, the symbiosis between dataflow and von Neumann architectures was tried to find by a number of research projects developing von Neumann/dataflow hybrids.

In this paper, several techniques (as well as machines based on them) for combining control-flow and dataflow will be described. In Section 2 we describe basics of hybrid dataflow computing. Selected hybrid architectures are given in Section 3. In Section 4 we describe the so-called micro dataflow which is nowadays used in state-of-the-art microprocessors. Finally, in Section 5 we compare the hybrid dataflow approaches and discusse some possible research directions.

2. Hybrid Dataflow

Key features of von Neumann/dataflow hybrids are given in Table 1.

The spectrum of such hybrids is quite broad, ranging from simple extensions of a von Neumann processor with a few additional instructions to specialized dataflow systems which attempt to reduce overhead by increasing the execution grain size and employing various scheduling, allocation, and resource management techniques developed for von Neumann computers. The results of these projects show that dataflow and von Neumann computers do not necessarily represent two entirely disjoint worlds but rather two extreme ends of a spectrum of possible computer systems (Beck et al., 1993 and Šilc et al., 1998).

In the following section we describe some basic terms and concepts, e.g., threaded dataflow, coarse-grain dataflow, complex dataflow, and RISC dataflow.

2.1. Threaded Dataflow

By the term *threaded dataflow* we understand a technique where the dataflow principle is modified so that instructions of certain instruction streams are processed in succeeding machine cycles. In particular, given a dataflow graph (program), each subgraph that exhibits a low degree of parallelism is identified and transformed into a sequential thread of instructions. Such a thread 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 used in 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 of the same thread is stored in registers instead of written back to memory. These registers may be referenced by any succeeding instruction in the thread. This improves single-thread performance because the total number of tokens needed to schedule program instructions is reduced which in turn saves hardware resources. In addition, pipeline bubbles are avoided for dyadic instructions within a thread.

Two threaded dataflow execution techniques can be distinguished – *direct token recycling* and *consecutive execution* of the instructions of a single thread. The first technique, direct token recycling, is used in the Monsoon dataflow computer. It allows a particular thread to occupy only a pipeline frame slot in the 8-stage pipeline. This 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. The second technique, consecutive execution, is used in Epsilon-2 and EM-4 which consecutively execute instructions from a thread. The circular pipelined machine organization 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 matching of further tokens in favor of consecutive issuing of all instructions of the started thread. For example, in *strongly connected arc* model, each arc of the dataflow graph is classified as either a normal arc or a strongly connected arc. The set of nodes that are connected by strongly connected arcs is called *strongly connected block*. While the standard instruction firing rule is that a node (instruction) is fired when all input arcs have matching tokens (operands), the enhancement for strongly connected blocks is that such a block is fired if its source nodes are enabled and the execution of the whole block is conducted as a unit, i.e., without applying the standard dataflow firing rule for other nodes in that block.

In all threaded dataflow machines, the central design problem is the implementation of an efficient synchronization mechanism (Sakai, 1995). *Direct matching* is a synchronization mechanism that needs no associative mechanisms. As in *explicit token store*, a matching area in operand memory is exclusively reserved for a single function instance. This area is called operand segment. The code block in instruction memory corresponding to operand segment is called a template segment. Its top address is called template segment number. Operand segment number points to the top of the operand segment. A token comprises an operand, operand segment number, a displacement, and a synchronization flag. Displacement serves both as a displacement of the destination instruction in the instruction memory and as a displacement of the matching operand in operand memory. Sinchronization flag indicates the type of synchronization which can be either monadic, left dyadic, right dyadic, and immediate. The matching address is produced by concatenating operand segment number and displacement. Instruction address is derived by concatenating template segment number and the displacement. Each slot in an operand segment also has a presence bit. A dyadic matching is performed by a test-and-set of the presence bit. If the presence bit has already been set, the partner data will be read, the bit will be cleared, and the instruction will be executed. Otherwise, the arriving data will be stored there and the presence bit will be set.

2.2. Coarse-Grain Dataflow

Coarse-grain dataflow is a technique for combining dataflow with control-flow which advocates activating macro dataflow *actors* in the dataflow manner while executing instruction sequences, represented by actors, in the von Neumann style. Coarse-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 to support 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.

2.3. Complex Dataflow

Another technique to reduce the overhead of the instruction-level synchronization 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 fashion, 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.

An advantage of complex machine operations is the ability to exploit parallelism at the subinstruction level. Therefore, such a machine has to partition a 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 FIFO-buffers 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 Sto*ll*mann Dataflow Machine, and the ASTOR architecture. These architectures combine complex machine instructions with coarse-grain dataflow, described above. The structure-flow technique proposed for the SIGMA-1 enhances these fine-grain dataflow computer by structure load/store instructions that can move, for instance, whole vectors to/from structure store. Arithmetic operations are executed by the cyclic pipeline within a PE.

2.4. RISC Dataflow

The development of *RISC dataflow* was an additional stimulus for dataflow/von Neumann hybrids. RISC dataflow architectures support the execution of existing software written for conventional processors. Using such a machine as a bridge between existing systems and new dataflow supercomputers should have made the transition from imperative von Neumann languages to dataflow languages easier to the programmer. The basic philosophy underlying the development of the RISC dataflow architecture was to use a RISC-like instruction set, to change the architecture to support multithreaded computation, to add fork and join instructions in order to manage multiple threads, to implement all global storage as I-structure storage, and to implement load/store instructions to execute in split-phase mode.

3. Selected Representatives

Next, we briefly describe some highly influential hybrid dataflow projects.

3.1. Threaded Dataflow

3.1.1. EM-4 and EM-X

In the EM-4 project (Sakai et al., 1989) the essential elements of a dynamic dataflow architecture using frame storage for local variables are incorporated into a single chip processor. In this design each *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 PEs. Since May 1990, the EM-4 prototype with 80 PEs is operational. Each PE consists of a processor (EMC-R) and a data repository obeying the single-assignment rule, called I-structure memory.

The processor and its memory (containing operand segments and template segments) are interfaced with memory control unit. The instruction buffer is used as a token store. A FIFO type buffer is implemented using a dual port RAM on chip. If this buffer is full, a part of the off-chip memory is used as secondary buffer. The fetch/matching unit is used for matching tokens and fetching instructions. It performs direct matching for packets and instruction sequencing for a strongly connected block (thread). The heart of the EMC-R is the execution unit, which fetches instructions until the end of the thread (if the next instruction is strongly connected with the current instruction, instruction fetch and data load of the next instruction are overlapped with the execution). Instructions with matching tokens are executed. Instructions can emit tokens or write to register file.

In 1993, an upgrade to EM-4, called EM-X, was developed (Kodama et al., 1995). It was designed to support latency reduction by fusing the communication pipeline with the execution pipeline, latency hiding via multithreading, and run-time latency minimization for remote memory access. EM-4 can access remote memory by invoking packet system handlers on the destination PE. Clearly, when the destination PE is busy, remote memory requests are blocked by the current thread execution. To remedy this, EM-X supports direct remote memory read/write mechanism, which can access the memory independently of thread execution. For these reasons, the EMC-Y single-chip processor was used in EM-X (instead of EMC-R that was used in EM-4).

3.1.2. Monsoon

The Monsoon dataflow multiprocessor (Papadopoulus and Culler, 1990) was built jointly by MIT and Motorola. In Monsoon, dataflow PEs are coupled with each other and with I-structure storage units by a multistage packet-switching network. Each PE is using an eight-stage pipeline. The first stage is the instruction fetch stage which precedes token matching (in contrast to dynamic dataflow processors with associative matching units). Such a 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 from the frame address and operand offset. This is done in the second stage, called effective address generation, which is the first of three pipeline stages that perform the token matching. In the third stage, called presence bit operation, a presence bit is accessed to find out if the first operand of a dyadic operation has already arrived. If not, the presence bit is set and the current token is stored into the frame slot of the frame memory. Otherwise, the presence bit is reset and the operand is retrieved from the slot. Operand storing or retrieving is the task of the fourth pipeline stage – frame operation stage. The next three stages are execution stages in which, besides other things, the next tag is computed concurrently. The eighth stage, also called form-token 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.

The Monsoon dataflow processor (Papadopoulos and Traub, 1991) can be viewed as a cycle-by-cycle interleaving multithreaded computer due to its ability of direct token recycling. Using this technique, a successor token is directly fed back in the eight-stage 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 digressing from the fine-grain dataflow execution model.

Since September 1990, 1 PE $\times$ 1 I-structure memory configuration (also referred to as the two-node system) is operational while the first 8 $\times$ 8 configuration (16-node system) was delivered in the fall of 1991. In total, sixteen two-node Monsoon systems were constructed and delivered to universities across the USA and two 16-node systems were delivered to MIT and Los Alamos National Laboratories.

3.1.3. Epsilon-2

Epsilon-2 machine (Grafe and Hoch, 1990) 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 a structure unit, connected via a 4 $\times$ 4 crossbar to each other, an I/O port, and the global interconnection network. 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.

3.1.4. RWC-1

The massively parallel computer RWC-1 (Sakai et al., 1993) is a descendant of EM-4 (as it is EM-X). A *multidimensional directed cycles ensemble* network connects up to 1 024 PEs. Two small-scale systems, Testbed-I with 64 PEs and Testbed-II with 128 PEs are used for testing and software development. The PE is based on *reduced interprocessor-communication architecture* which employs 2-issue superscalar execution, a floating-point multiplier/adder module, and offers fast and simple message handling mechanism, hard-wired queuing and scheduling mechanism, a hard-wired micro-synchronization mechanism, integration of communication, scheduling and execution, and simplification of the integrated structure (Matsuoka et al., 1998).

3.2. Coarse-Grain Dataflow

3.2.1. StarT

The StarT project was launched by MIT and Motorola in mid-1991. The StarT, sometimes also written as *T (Nikhil et al., 1992), is a direct descendant of dataflow architectures, especially of the Monsoon, and unifies them with von Neumann architectures. StarT 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 StarT node consists of the *data processor* (dP), which executes threads, the *synchronization coprocessor* (sP), which handles returning load responses and join operations, and the *remote-memory request processor* (RMem) for incoming remote load/store requests. The three components share local *node memory*. The node is coupled with a high performance network having a fat-tree topology with high cross-section bandwidth.

Due to its on-chip special-function unit, the 2-issue superscalar RISC microprocessor Motorola 88110 was chosen as the basis for the node implementation. However, in order to keep the communication latency to a minimum, a number of logic modules were added to the 88110 chip to make it acting as a tightly-coupled network interface. The resulting chip was called 88110MP (MP for multiprocessor) with 10 - 20 machine cycles overhead for sending and receiving data between the node and the network. Two 88110MP microprocessors were used to implement the StarT node. The first one operated as dP, with its special-function unit serving as sP. dP and sP were optimized for long and short threads, respectively. The second 88110MP was tailored to act as RMem to handle remote memory requests from other nodes to the local node memory (64 MB).

The fat-tree network was based on the MIT Arctic packed routing chip (Boughton, 1994) that was twice as fast as Monsoon's PaRC and was expected to drive the interconnection network at 1.6 Gbyte/s/link in each direction with packet sizes ranging from 16 to 96 bytes. Sixteen nodes were packaged into a "brick" with 3.2 GFLOPS and 3 200 MIPS peak performance. Sixteen bricks can be interconnected into 256-node machine with the potential to achieve 50 GFLOPS and 50 000 MIPS.

As reported in (Arvind et al., 1997), MIT decided to go back to the drawing board and to start afresh on PowerPC-based StarT machines after Motorola and IBM started manufacturing PowerPC family of RISC microprocessors. Thus, PowerPC 620 was planned in StarT-ng machine (Ang et al., 1995) but the architecture was redesigned once again – this time around a 32-bit PowerPC 604 – and was called StarT-Voyager machine (Ang et al., 1996). This machine, however, bears little resemblance to the original StarT architecture and no similarity to Monsoon.

3.2.2. TAM

The Threaded Abstract Machine (TAM) (Culler et al., 1991) 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.

3.2.3. ADARC

In the Associative Dataflow Architecture (ADARC) the processing units are connected via an associative communication network (Strohschneider et al., 1994). 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 associative switching 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.

3.2.4. Pebbles

The Pebbles architecture (Roh and Najjar, 1995) is a coarse-grain dataflow architecture with a decoupling of the synchronization unit and the execution unit within the PEs. The PEs 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-issue superscalar microprocessor.

3.2.5. MTA and EARTH

The Efficient Architecture of Running Threads (EARTH) (Hum et al., 1994 and Maquelin, 1995) is based on the MTA (Multithreaded Architecture) 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's local memory, which is cached. Accessing data in a remote processor requires explicit request and sends 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. MTA or EARTH rely on non-blocking threads. The EARTH architecture is implemented on top of the experimental (but rather conventional) MANNA multiprocessor.

3.3. Complex Dataflow

3.3.1. ASTOR

The Augsburg Structure-Oriented Architecture (ASTOR) (Zehendner and Ungerer, 1987) 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 dependence construct describes the partial order in the execution of instructions. It can be visualized by a dependence graph. The nodes in a dependence graph represent *control constructs* or *data instructions*; the directed arcs denote control dependences between the nodes. Tokens are propagated along the arcs of the dependence graph. To distinguish different activations of a dependence graph, a tag is assigned to each token. The firing rule of dynamic dataflow is applied but tokens do not carry data.

The ASTOR architecture consists of PEs connected by an *instruction communication network* to transfer procedure calls and *data communication network* for parameter passing. No global storage is used. Due to the separation of code and data objects, each PE consists of two loosely coupled parts: First, the *program flow control part* consists of a static and dynamic code storage, the static and the dynamic code access manager, the I/O managers, and the control construct managers (individualy named call, loop, choice and dependency manager). Second, the *data object processing part* consists of a data storage, several data access managers, an I/O manager, some data transformation units, and the computational structure manager. All managers in a PE work in parallel to each other. Asynchronous processing and decoupling of the managers is achieved by buffering the links between them.

3.3.2. Sto*ll*man Dataflow Machine

The Stollman dataflow machine (Glück-Hiltrop et al., 1989) 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 SQL) is viewed as a 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.

3.3.3. DGC

In Decoupled Graph/Computation (DGC) architecture (Evripidou and Gaudiot, 1991) the token matching and token formatting and routing are reduced to a single graph operation called *determine executability*. 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).

3.4. RISC Dataflow

3.4.1. P-RISC Architecture

The Parallel RISC (P-RISC) architecture (Nikhil and Arvind, 1989) based on the above principles and consists of a collection of PEs (with local memory) and a *global memory*, interconnected through a packed-switching communication network.

Following the principles underlying all RISC architectures, the ALU of P-RISC PEs distinguishes between *load/store* instructions, which are the only instructions accessing global memory (implemented as I-structure storage), and *arithmetic/logical* instructions, which operate on local memory (registers). Fixed instruction length and one-cycle instruction execution (except for load/store instructions) are the characteristics of this processor. In addition, P-RISC lacks any explicit matching unit. Instead, all operands associated with a sequential thread of computation are kept in a frame in local program memory. Each execution step makes use of an $\langle$ IP, FP $\rangle$ pair, where IP serves to fetch the next instruction while FP serves as the base for fetching and storing operands. The pair is called *continuation* and corresponds to the tagged part of a token in a tagged-token dataflow machine. To make P-RISC multithreaded, the stack of frames must be changed to a tree of frames, and a separate continuation must be associated with each thread. The frame tree allows different threads of instructions accessing different branches of the tree concurrently while the separate continuation extents the concept of a single PC and a single operand base register to multiple instances. Continuations of all active threads are held in the continuation queue. At each clock cycle, a continuation (also called token) is dequeued and inserted into the pipeline. It is first processed by the instruction fetch unit, which fetches from instruction memory the instruction pointed to by IP. Next, operands are fetched from program memory by the operand fetch unit. This uses operand offsets (specified in the instruction) relative to the FP. The executable token is passed to the ALU or, in case of a load/store instruction, to the global memory. To solve the memory latency problem, the load/store instructions are implemented to operate in a split-phase manner. The execution of an ALU instruction produces result tokens and new continuations. Result tokens are stored in the appropriate frame in (local) frame memory by the operand store unit. Continuations are new $\langle$ FP, IP $\rangle$-pairs, generated by incrementing the current IP value or, in case of a branch instruction, replacing it by the target pointer. They are enqueued in the continuation queue of the local PE.

4. Micro Dataflow

In addition, the latest generation of microprocessors – as exemplified by the Intel Pentium III, MIPS R12000, Hewlett-Packard PA-8500, DEC Alpha 21264, etc. – absorbed some ideas from the dataflow approach.

In particular, state-of-the-art processors usually display an out-of-order dynamic execution whereby the processor dynamically issues an instruction as soon as all its operands are available and the required execution unit is not busy. This technique is referred to as local dataflow or *micro dataflow* by microprocessor researchers (Šilc et al., 1999).

In the first paper on the Pentium Pro the instruction pipeline is decribed as follows (Colwell and Steck, 1995): *"The flow of the Intel Architecture instructions is predicted and these instructions are decoded into micro-operations (μops), or series of ops, and these 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."* That is, after a branch prediction (to remove control dependences) and register renaming (to remove antidependences and output dependences) the instructions (or ops) are placed in the *instruction window* of pending instructions, where ops are executed in the dataflow fashion, and then in a reorder buffer that restores the program order and execution states of the instructions. Instruction window and reorder buffer may coincide. State-of-the-art microprocessors typically provide 32 (in MIPS R10000), 40 (in Intel Pentium Pro) or 56 (in HP PA-8000) instruction slots in the instruction window or reorder buffer. Each instruction is ready to be executed as soon as all operands are available. A 4-issue superscalar processor issues up to 4 executable instructions per cycle to the execution units, provided that resource conflicts do not occur. Issue and execution determine the out-of-order section of a microprocessor. After execution, instructions are retired in program order.

Comparing advanced dataflow computers with such superscalar microprocessors reveals several similarities as well as 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 today's microprocessors, dataflow approach resolves any threads of control into separate instructions that are ready to execute as soon as all required operands become available. Thus, the *fine-grained parallelism potentially utilized* by a dataflow computer is far larger than the parallelism available for microprocessors.

Data and control dependences potentially cause pipeline hazards in microprocessors that are handled by complex forwarding logic. In fine-grain dataflow computers pipeline hazards are avoided due to the continuous context switches (with the disadvantage of a poor single thread performance). In microprocessors, antidependences and output dependences 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 antidependences and output dependences 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 superscalar microprocessor is *branch prediction and speculative execution*. In microrocessors, the accuracy of the branch prediction is surprisingly high – more than 95 % are reported in (Chang et al., 1994) for single SPEC benchmark programs. However, rerolling execution in case of a wrongly predicted path is costly in terms of processor cycles, especially in deeply pipelined microprocessors. In the dataflow environment, however, the idea of branch prediction and speculative execution has *never* been evaluated. The reason for this may be that dataflow was considered to produce an abundance of parallelism while speculation leads to speculative parallelism which is – because of instruction discarding when branch is mispredicted – inferior to "real" parallelism.

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 operand *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 become 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 likelihood of the arrival of a matching token is difficult to estimate so caching of tokens to be matched soon is difficult in dataflow.

A large instruction window is crucial for today's and future superscalar microprocessors to find enough instructions for parallel execution. However, the control logic for very large instruction windows gets so complex that it hinders higher cycle rates. Therefore alternative instruction window organizations are needed. In (Palacharla et al., 1997) a multiple FIFO-based organization is proposed. Only the instructions at the heads of a number of FIFO buffers can be issued to the execution units in the next cycle. The total parallelism in the instruction window is restricted in favor of a less costly issue 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 organizations, 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 issues very small compiler-generated code sequences serially (in an otherwise fine-grained dataflow computer). Transferred to the local dataflow in an instruction window, an issue string might be used where a series of data dependent instructions are generated by a compiler and issued serially after the issue of the leading instruction. However, the high number of speculative instructions in the instruction window remains.

5. Comparison and Discussion

The architectures that were described in this paper are only a part of a broader spectrum of architectures, with von Neumann approach at one end and fine-grain dataflow approach on the other end (Table 2).

Table 3 compares these architectures according to several attributes, such as the type of instruction execution parallelism, and the type of synchronization mechanism.

As already stated by Maurice V. Wilkes in his book *Computing Perspectives*, if any practical machine based on dataflow ideas and offering real power emerges, it will be very different from what the originators of the concept had

Architecture	Key features
threaded dataflow	(The closest architure type to the fine-grain dataflow machine.) instructions of certain instruction threads are processed in succeeding machine cycles
coarse-grain dataflow	activates macro dataflow actors in the dataflow manner but uses control-flow to execute each of these actors
complex dataflow	complex machine operations can be implemented by pipeline techniques
RISC dataflow	parallel control operators based computational model (The closest architure type to the von Neumann machine.)

Table 2. Key features of hybrid dataflow architectures.

Architecture	Parallelism	Execution mode	Synchronization	Matching	Registers
von Neumann	one control thread		none	no	yes
multithreaded von Neumann	several active control threads		full/empty bits locked	no	yes
RISC dataflow	several active control threads	vector pipelining	control operators	yes	yes
coarse-grain dataflow	several active control threads	vector pipelining	matching operations	yes	yes
complexw dataflow	several active control threads with complex machine operatin	vector pipelining	matching operations	yes	yes
threaded dataflow	several active control threads	repeat-on-input	matching operations	yes	yes
fine-grain dataflow	single instructions		matching operations	yes	no

Table 3. Augmenting dataflow with control-flow.

in might. We have shown in this paper that due to implementation problems fine-grain dataflow remains to be a simple and elegant theoretical model.

Dataflow machines, and especially fine-grain dataflow machines, are no longer considered to be a viable option for general purpose computations. For DSP algorithms however, the dataflow model of architecture is a natural fit. Especially the coarse-grain dataflow machines overcome many of the problems encountered in fine-grain dataflow architectures. The coarse-grain dataflow architectures enable efficient implementations for high performance digital signal processing.

It seems that, in order to build real dataflow machines, dataflow has to borrow some concepts from the von Neumann model of computation. On the other hand, the research in modern microprocessor architecture revealed the fruitfulness of dataflow conceps in the use of instruction level parallelism.

As a result, the dataflow and control-flow research communities now study many of the same questions. In principle, an algorithm defines a partial ordering of instructions due to control and data dependences. The total ordering in an instruction stream for today's microprocessors stems from von Neumann languages. But why should

- a *programmer*
  - design a *partially* ordered algorithm,
  - and then 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 then 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 re-establish the unnatural serial program order for in-order commitment in the retire 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 issue out of the instruction window. The processor dismisses the serial reordering in the completion stage in favor of only a partial reordering. The retire 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 one thread does not restrict retirement of instructions of another thread.

References

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

[2] B.S. Ang, D. Chiou, L. Rudolph, Arvind, Message passing support on StarT-Voyager. Technical Report MIT/CSG Memo 387, Laboratory for Computer Science, MIT, Cambridge, 1996.

[3] Arvind, L. Bic, T. Ungerer, Evolution of dataflow computers. In *Advanced Topics in Data-Flow Computing* (J.-L. Gaudiot, L. Bic, Eds.) (1991) pp. 3–33. Prentice Hall, Engelewood Cliffs.

[4] Arvind, A.T. Dahbura, A. Caro (1997), Computer architecture research and the real world. Technical Report MIT/CSG Memo 397, Laboratory for Computer Science, MIT, Cambridge, 1997.

[5] M. Beck, T. Ungerer, E. Zehender, Classification and performance evaluation of hybrid dataflow techniques with respect to matrix multiplication. Presented at the *Proceedings of the GI/ITG Workshop PARS*, (1993) pp. 118–126, Dresden, Germany.

[6] G.A. Boughton, Arctic routing chip. *Lect. Notes Comput. Sc.*, **853** (1994),310–317.

[7] P.-Y. Chang, E. Hao, T.-Y. Yeh, Y.N. Patt, Branch classification: A new mechanism for improving branch predictor performance. Presented at the *Proceedings of the 27th International Symposium on Microarchitecture*, (1994) pp. 22–31, San Jose, CA.

[8] R.P. Colwell, R.L. Steck, A 0.6 m BiCMOS processor with dynamic execution. Presented at the *Proceedings of the International Solid State Circuits Conference*, (1995) pp. 176–177.

[9] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken, J. Wawrzynek, Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. Presented at the *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, (1991) pp. 164–175, Santa Clara, CA.

[10] P. Evripidou, J.-L. Gaudiot, The USC decoupled multilevel data-flow execution model. In *Advanced Topics in Data-Flow Computing* (J.-L. Gaudiot, L. Bic, Eds.) (1991) pp. 347–379. Prentice Hall, Englewood Cliffs.

[11] E. Glück-Hiltrop, M. Ramlow, U. Schürfeld, The Stollman dataflow machine. *Lect. Notes Comput. Sc.*, **365** (1989),433–457.

[12] V.G. Grafe, J.E. Hoch, The Epsilon-2 multiprocessor system. *J. Parall. Distr. Comput.*, **10** (1990),309–318.

[13] H.H.J. Hum, K.B. Theobald, G.R. Gao (1994), Building multithreaded architectures with off-the-shelf microprocessor. Presented at the *Proceedings of the 8th International Parallel Processing Symposium*, (1994) pp. 288–294, Cancún, Mexico.

[14] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, Y. Yamaguchi Y, The EM-X parallel computer: Architecture and basic performance. Presented at the *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (1995) pp. 14–23, Santa Margherita Ligure, Italy.

[15] O.C. Maquelin, H.H.J. Hum, G.R. Gao, Costs and benefits of multithreading with off-the-shelf processors. *Lect. Notes Comput. Sc.*, **966** (1995),117–128.

[16] H. Matsuoka, K. Okamoto, H. Hirono, M. Sato, T. Yokota, S. Sakai, Pipeline design and enhancement for fast network message handling in RWC-1 multiprocessor. Presented at the *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation*, (1998), Las Vegas, NE.

[17] R.S. Nikhil, Arvind, Can dataflow subsume von Neumann computing? Presented at the *Proceedings of the 16th Annual Symposium on Computer Architecture*, (1989) pp. 262–272, Jerusalem, Israel.

[18] R.S. Nikhil, G.M. Papadopoulos, Arvind, *T: A multithreaded massively parallel architecture. Presented at the *Proceedings of the 19th Annual Symposium on Computer Architecture*, (1992) pp. 156–167, Golden Coast, Australia.

[19] S. Palacharla, N.P. Jouppi, J.E. Smith, Complexity-effective superscalar processors. Presented at the *Proceedings of the 24th Annual International Symposium on Computer Architecture*, (1997) pp.

206–218, Denver, CO.

[20] G.M. Papadopoulos, D.E. Culler, Monsoon: An explicit token-store architecture. Presented at the *Proceedings of the 17th Annual Symposium on Computer Architecture*, (1990) pp. 82–91, Seattle, WA.

[21] G.M. PAPADOPOULOS, K.R. TRAUB, Multithreading: A revisionist view of dataflow architectures. Presented at the *Proceedings of the 18th Annual Symposium on Computer Architecture*, (1991) pp. 342–351, Toronto, Canada.

[22] L. ROH, W.A. NAJJAR, Design of storage hierarchy in multithreaded architectures. Presented at the *Proceedings of the 28th International Symposium on Microarchitecture*, (1995) pp. 271–278, Ann Arbo, MI.

[23] S. SAKAI, Synchronization and pipeline design for a multithreaded massively parallel computer. In *Advanced Topics in Dataflow Computing and Multithreading* (G.R. GAO, L. BIC, J.-L. GAUDIOT, Eds.) (1995) pp. 55–74. IEEE Computer Society Press, Los Alamitos.

[24] S. SAKAI, K. OKAMOTO, K. MATSUOKA, H. HIRONO, Y. KODAMA, M. SATO, T. YOKOTA, Basic features of a massively parallel computer RWC-1. Presented at the *Proceedings of the 1993 Joint Symposium on Parallel Processing*, (1993), Tokyo, Japan.

[25] S. SAKAI, Y. YAMAGUCHI, K. HIRAKI, Y. KODAMA, T. YUBA, An architecture of a dataflow single chip processor. Presented at the *Proceedings of the 16th Annual Symposium on Computer Architecture*, (1989) pp. 46–53, Jerusalem, Israel.

[26] J. ŠILC, B. ROBIČ, T. UNGERER, Asynchrony in parallel computing: From dataflow to multithreading. *Parall. Distr. Comput. Practices*, **1** (1998),57–83.

[27] J. ŠILC, B. ROBIČ, T. UNGERER, *Processor Architecture – From Dataflow to Superscalar and Beyond*. Springer-Verlag, Heidelberg, Berlin, New York 1999.

[28] J. STROHSCHNEIDER, B. KLAUER, S. ZICKENHEIMER, K. WALDSCHMIDT, Adarc: A fine grain dataflow architecture with associative communication network. Presented at the *Proceedings of the 20th Euromicro Conference: System Architecture and Integration*, (1994) pp.445–450, Liverpool, England.

[29] E. ZEHENDNER, T. UNGERER, The ASTOR architecture. Presented at the *Proceedings of the 7th International Conference on Distributed Computing Systems*, (1987) pp. 424–430, Berlin, Germany.

Received: February, 1999
Revised: February, 2000
Accepted: March, 2000

Contact address:
Borut Robič
Faculty of Computer and Information Science
University of Ljubljana
Tržaška 25
1001 Ljubljana, Slovenia
phone: +386-61-176 8256
e-mail: `borut.robic@fri.uni-lj.si`

Jurij Šilc
Computer Systems Department
Jožef Stefan Institute
Jamova 39
1001 Ljubljana, Slovenia
phone: +386-61-177 3268
e-mail: `jurij.silc@ijs.si`

Theo Ungerer
Department of Computer Design and Fault Tolerance
University of Karlsruhe
76 128 Karlsruhe, Germany
phone: +49-721-608 6048
e-mail: `ungerer@ira.uka.de`

BORUT ROBIČ is an associate professor of Computer Science at the University of Ljubljana, Slovenia. From 1984 to 1993 he has been assistant researcher at the Department of Computer Science and Informatics, Jožef Stefan Institute, Ljubljana. From 1994 to 2000 he has been a researcher at the Computer Systems Department at the same institute, and also an assistant professor at the Faculty of Computer and Information Science, University of Ljubljana. Robič received the Ph.D. degree in computer science from University of Ljubljana, Slovenia, 1993. His present research interests include parallel computating, computational and complexity theory, and algorithm design.

JURIJ ŠILC is a researcher at the Jožeef Stefan Institute in Ljubljana, Slovenia. From 1980 to 1986 he has been assistant researcher at the Department for Computer Science and Informatics. From 1987 to 1993 he has been the Head of the Laboratory for Computer Architecture at the same department. Since 1994 he has been Deputy Head of the Computer Systems Department at the Jožef Stefan Institute. Šilc received his Ph.D. degree in Electrical Engineering from University of Ljubljana, Slovenia, in 1992. His research interests include parallel computation, computer architecture and high-level synthesis.

THEO UNGERER is a professor of Computer Science at the University of Karlsruhe, Germany. Previously, he was scientific assistant at the University of Augsburg (1982-89 and 1990-92), visiting assistant professor at the University of California, Irvine (1998-90), professor of computer architecture at the University of Jena, Germany (1992-1993). Since 1993 he is with the Department of Computer Design and Fault Tolerance, University of Karlsruhe. Ungerer received a Doctoral Degree at the University of Augsburg in 1986, and a second Doctoral Degree (Habilitation) at the University of Augsburg in 1992. His current research interests are in the area of processor architecture and the area of parallel and distributed computing. He is a member of the ACM, the IEEE, the GI, and FIFF.