ADAM — An Abstract Dataflow Machine and Its Transputer Implementation

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

February 13, 1996

Abstract

This paper describes the design of the ADAM Abstract Dataflow Machine. This abstract machine has a hybrid distributed-memory architecture combining features from conventional register machines and from dataflow computers. The ADAM processor has a conventional instruction set augmented by primitives for the creation and synchronization of parallel tasks. Memory units on each ADAM module are responsible for the management of non-strict data structures. A prototype implementation of the concept on a multi-transputer system shows promising speed-ups.

1 Introduction

In the last decade, there has been much international research on how to efficiently exploit the fine-grained parallelism inherent in all algorithms. Non-von Neumann architectures have been designed in order to effectively utilize this parallelism by hardware primitives that minimize the overhead for the creation and synchronization of large numbers of parallel tasks. Dataflow architectures seem to be among the most promising of these approaches. Several dataflow computers have already been realized in hardware, e.g. the *Manchester Dataflow Computer* [1], the *ETL SIGMA-1* at the Japanese Electrotechnical Laboratory [2] and the *ETS Explicit Token Store "Monsoon"* at the MIT [4].

In spite of these efforts, commercially available dataflow systems have not yet come into existence. One of the main reasons for this disappointing fact is the enormous amount of knowledge gathered on the design and use of conventional processors, as well from the hardware engineer's as from the software engineer's point of view. Recently, there have therefore been several attempts to combine the benefits of both worlds, of data-driven and of control-driven architectures [3].

In this paper, the main results of the author's diploma thesis [5] are presented dealing with the design and the implementation of a dataflow simulator on conventional parallel distributed memory hardware (i.e. on a multi-transputer system). The dataflow style

Figure 1: A Dataflow Graph

of programming shows essential benefits that have no counterpart in classical parallel programming. Our work is intended to show that this paradigma can also be based on conventional control-driven hardware. This is achieved by the ADAM Abstract Dataflow Machine that simulates dataflow behaviour by control-driven program execution.

2 Basic Idea

The basic idea of our approach is as follows. We take a mathematical expression, e.g.

$$(a+b)*(-y) \ \textbf{where}$$
$$a = -x$$
$$b = x/y$$

This expression can be viewed as the body of a function $f(x, y)$ with formal parameters x and y and is translated in a dataflow graph as depicted in Fig. 1.

In dataflow machines, this graph is represented by a program similiar to the following one where each instruction consists of an operation code and the address of its successor:

```
0:  copy 2 3.l
1:  copy 3.r 4
2:  neg 5.l
3:  div 5.r
4:  neg 6
5:  add 6
6:  mul 7
```

Each node of the graph is translated into one instruction of the program where the arcs of the graph are represented by the instructions' successor addresses. Branches are translated into explicit `copy` instructions duplicating their input values. If the successor is a binary instruction, the predecessor must also specify which of the "input ports" of the successor is referenced.

In dataflow machines, values are transmitted between different instructions by pieces of information, called "tokens", that flow along the arcs of the dataflow graph. For instructions with more than one input arcs, special mechanisms have to handle the synchronization of the input tokens, i.e. a "rendezvous" between the partner tokens has to be organized.

In the ADAM model, the arcs of the dataflow graph denote registers and the nodes represent two-address instructions operating on these registers:

0:	copy	x_1, x_2, 2
1:	copy	y_1, y_2, 4
2:	split	3, 6
3:	neg	x_1, x_1, 8
4:	split	5, 6
5:	neg	y_2, y_2, 10
6:	wait	e_0, 7
7:	div	x_2, y_1, 8
8:	wait	e_1, 9
9:	add	x_1, x_2, 10
10:	wait	e_2, 11
11:	div	x_1, y_2, 12

Each instruction operates on two registers where the first one receives the result value. After an instruction has been executed, control flow is routed by the address field to its successor. `split` instructions fork the actual thread of control into two parallel threads starting at the specified successors. In above program, two threads enter the code block at addresses 0 and 1. The execution of each thread may be unhinderedly continued as long as no `wait` instruction is encountered.

Each `wait` guards a binary *event* encoded in the first field of this instruction. Each event may have one of the values "occurred" or "not occurred", initially no event has occurred. If execution reaches a `wait` whose event has not yet occurred, the actual thread of control is interrupted and the referred event is set to "occurred". The next time, this event is tested, the actual thread may unhinderedly pass the `wait`.

`split` and `wait` control the generation of parallel processes and the synchronization between them, they are the only instructions related to parallelism. However, they suffice to control the program execution in such a way that dataflow semantics is simulated.

We want to emphasize the essential difference between the pure dataflow model and our approach: In dataflow architectures, all data physically "flow" between instructions and, by this, control the computation. In the ADAM processor, threads of instructions are executed without any data flowing between them. Communication between different instructions is done via shared registers, synchronization between parallel threads are explicitely handled by `wait` statements.

This approach has two essential advantages:

1. In dataflow machines, all data referenced by more than one instruction have to be **explicitly duplicated** and sent to the required instructions, even if they belong to the same activation of a function and reside on the same processor. In our model, parallel program paths may **share data** held in common registers, an explicit duplication can be avoided in most cases.

2. In dataflow architectures, the synchronization model is fixed in hardware. Parallel program paths have to be generated and finally synchronized even if parallelism is not really utilized. This is especially the case if only whole function activations are placed on new processors i.e. parallel paths within an activation are always executed on the same processor. In our approach, a compiler is able to detect in which cases parallel paths may be safely (i.e. without changes to the dataflow semantics) executed in sequential and **serialize the code**[1].

Using both optimizations, we may rewrite the above program as follows:

0:	wait	e_0, 1
1:	copy	x_2, x_1, 2
2:	div	x_2, y_1, 3
3:	neg	x_1, x_1, 4
4:	add	x_1, x_2, 5
5:	neg	y_1, y_1, 6
6:	mult	x_1, y_1, 7

Assuming that there are two parallel tasks delivering their arguments in registers x_1 and y_1, there is only one `wait` instruction required to synchronizes these tasks. All other instructions may be sequentially executed requiring only one additional register x_2. Hence, the efficiency of the function body drastically increases, while the overall dataflow-like behaviour of a program consisting of a set of such functions remains unchanged.

3 Global Architecture

The ADAM architecture consists of several equally constructed worker modules that are connected by a network of unbuffered uni-directional channels. One special module, the *master*, is connected to the peripheral devices for input and ouput. The master handles all interactions with the outside world, in particular, it is responsible for loading and distributing the program to be executed, for triggering the execution of the program and for delivering the result values of the computation.

Each worker module of the ADAM consists of three components illustrated in Fig. 2:

- **The Arbiter** is the only unit with an interface to the neighbour modules. It receives messages from the other local units and from the neighbour modules and routes them to their correct places of destination.

[1] Sequentialization is not allowed in the case of non-strict functions, i.e. functions that may return values even if not all arguments have yet arrived. In general, a process called *strictness analysis* has to decide this question.

Figure 2: An ADAM Worker Module

- **The Processor Unit** executes the given program and, thereby, performs the actual computation. It receives messages (basically arguments for the local function activations) from and sends messages (basically results of local function activations) to other processors in the network.

- **The Memory Unit** contains a heap-organized store holding tuples, i.e. data structures of fixed size. Tuples may be allocated and referenced by a (local or distinct) processor via messages to the memory unit. Result values (either atomic values or tuple references) are sent as messages back to the callers.

Each worker module has only a restricted view of the network. It knows about the addresses and certain state parameters of its neighbours but has in general no global information about the network. The only exception is the arbiter that holds a network routing table in order to forward incoming messages to their correct destination.

4 The Processor Unit

The ADAM processor is basically a conventional control-driven processor with several extensions that allow the efficient simulation of dataflow behaviour. It consists of the following components (depicted in Fig. 3):

- **The Execution Unit** performs the arithmetical/logical computations.

- **The Program Store** holds a sequence of *code blocks* each of which corresponds to one user-defined function. Every processor holds the whole program code and may therefore execute any function of the program.

Figure 3: The ADAM Processor

- **The Frame Store** is a heap-organized store holding a set of *activation frames* each of which corresponds to one particular function activation. Each frame represents the actual state of its activation by describing the contents of the registers and the state of the events.

- **The Context Store** is a table of activation descriptors called *contexts*. Each context uniquely describes one function activation on the processor by the base addresses of the activation's code block and of its register frame.

- **The Trigger Store** contains a set of *instruction triggers*, each of which is the description of a thread that may be started but has not yet been. When a `split` instruction is executed, a trigger for one branch is put into this store.

- **The Message Store** and **the Request Store** buffer all messages received. The first store holds arguments for local function activations and result values of parallel activations. The second store holds requests to activate new function instances.

The interaction between these components is illustrated in Fig. 4: When the execution of a new thread is started, the description of the actual activation (i.e. the base addresses of code block and register frame) is stored in two registers. Any register number encoded in an instruction is actually an offset to the frame pointer. Hence, every function activation has its private set of registers. All event numbers referenced by `wait` instructions are actually *negative* offsets to the frame base address. Hence, every activation has its private sets of events each of which is represented by a bit in the frame.

If a `split` instruction is executed, the start address of one of the parallel branches is pushed on the trigger store while the other one is continued by the current thread.

If a `wait` instruction fails, the actual thread is terminated. In this case, an instruction trigger is popped from the trigger store and a new thread (in the current activation) is

Figure 4: Interaction between Processor Components

started. Only if the trigger store is empty, a context switch has to be made and a new activation is executed. This may be triggered by one of two events:

1. A **message** is received from the message store. In this case, a new thread is started beginning with the instruction that represents the destination of the message. Such a message may either hold the argument for a function activation or it may hold the result value delivered by some activation or it may also hold the result of a memory request (see the next section).
2. An **activation request** is received from the request store. In this case, some neighbour processor has sent a request to activate a new instance of some particular function. If the processor decides to fulfill this request, a new activation frame and a new context descriptor are allocated and a reference to the context descriptor is sent back to the caller.

The decision between these two possibilities depends on the actual load of the processor. If it is heavily loaded, argument messages will be preferred, if the load is low, more activation requests will be fulfilled. This alternative is (besides the decision to which processor to send activation requests to) the basic tool for load balancing.

5 The Memory Unit

Since processor units are only able to handle atomic (integer) data, the management of compound data structures has been shifted to separate memory units. Each worker module contains one memory unit that is constructed as follows (see Fig. 5):

- **The Heap** holds a set of tuples, i.e. data structures of arbitrary but fixed size. Each field of a tuple is either empty or holds an integer atom or a tuple reference.

Figure 5: The ADAM Memory Unit

All heaps on all network modules represent a global store, i.e. all references denote unique tuples within the network.

- **The Memory Manager** handles incoming requests to allocate a new tuple (`alloc`), to write a value into some tuple field (`write`) or to return the contents of some tuple field (`read`).
- **The "Expensive" Store** and **the "Cheap" Store** hold all incoming requests classified according to the fact whether fulfilling the request may require the allocation of heap memory ("expensive") or not ("cheap").

All tuples are **non-strict** i.e. references to them may be used even if (some or all of) their fields do not yet contain valid data. As a consequence, one process may allocate a tuple and send the tuple reference to another process *before* it has written any value into the fields of the tuple! However, in order to preserve determinism of program execution, special synchronization tools have been introduced:

Every field of a tuple has associated a **full/empty bit** indicating if a value has yet been written into this field or not. When a new tuple is allocated, all fields are marked as empty. If a read request arrives for a field that has not yet been defined, this request is put into a waiting queue and a reference to this queue is stored in the field. If then a write request for this field arrives, all waiting read requests are dequeued and the desired values are returned to the senders. However, if an attempt is made to overwrite an already defined tuple field, an error is reported. Hence, all tuples obey the **single-assignment rule** which is essential for non-strict data structures in parallel systems.

From the point of view of the ADAM instruction set, the allocation instruction `alloc` and the read instruction `read` are non-blocking: they immediately return their results, i.e. a reference to a new tuple and the contents of some tuple field, respectively. However, `read` and `write` are actually **split-phase operations**: Having submitted the memory

Program	Input	1T / 2W Ring	2T / 2W Ring	4T / 4W Full	8T / 8W Chordal Ring	16T / 16W Torus
FIBO	20	54(1.0)	28(1.9)	16(3.3)	10(5.4)	7(7.7)
PRIMES0	2000	302(1.0)	163(1.8)	93(3.2)	59(5.1)	41(7.4)
PRIMES1	10000	237(1.0)	127(1.9)	88(2.7)	65(3.7)	51(4.6)

Figure 6: Several ADAM Benchmarks

requests, the processor immediately breaks the execution of the actual thread and starts another one. The interrupted thread is only continued at the arrival of the desired value.

This feature has a very important consequence on the performance of the system: since the processor does *not* have to wait for the result of a memory operation, the architecture is able to **tolerate extremely long memory latencies**. This is especially important in a distributed memory architecture, since any pointer may reference a tuple that is sited on another (even non-neighbour!) module of the network.

The ADAM architecture uses for garbage collection a **weighted reference counting algorithm**: Every tuple reference has associated a weight and every tuple has stored the sum of the weights of all pointers referencing the tuple. If this sum decreases to zero, the tuple may be reclaimed. The essential advantage of a reference counting method in contrast to other garbage collection algorithms is its suitability for distributed memory systems: garbage collection is done **during the normal computation**, all unreferenced cells are immediately reclaimed and no separate collection phase (eventually involving the whole network) is necessary.

The use of *weighted* references has the additional advantage that inter-processor communication is minimized: If a pointer is copied, both copies get half of the original weight, hence, no message to the involved tuple is necessary. Only if the weight decreases to zero, a memory request for a new reference is to be sent. In this case, the reference count of the tuple is increased (up to some upper bound considered as "infinity") and a pointer with the new weight is returned. With an eight bit reference count field and a two bit reference weight (storing the binary logarithm of the weight), the overall overhead for garbage collection is neglectible.

6 Implementation

Our work on the ADAM architecture is part of the project "ADAM & EVE" dealing with the prototype implementation of the non-strict functional programming language EVE on conventional parallel hardware using principles of dataflow architectures. The target hardware consists of a multiprocessor containing 16 INMOS T800 transputers that may be dynamically configured to the desired topology. At present, we have implemented the runtime system interpreting ADAM machine code (each transputer usually simulates one ADAM module) and a PC-based ADAM assembler generating this code.

Using this assembler, several test programs have been hand-coded and show rather promising speedups. Some of the results are listed in Fig. 6. Each column of the table represents one particular processor configuration with the number of transputers (T) used, the number of worker modules (W) simulated and the topology of the transputer network listed above. Each item in the table denotes the elapsed time in seconds, the values in parentheses denote the relative speedups achieved.

The performance of the implementation has been tested with the benchmark programs FIBO (a doubly recursive fibonacci number generator), PRIMES0 (a prime number generator using auxiliary lists) and PRIMES1 (a prime number generator making use of difference lists). The efficiency (speedup/processor ratio) ranges in both of the first test programs from more than 90 % to about 45 %. We are not yet able to estimate the influence of the processor topology on the results. The third test program shows significantly lower speedups, which is a consequence a not yet satisfactory load balancing.

7 Future Work

In the future, we will exhaustively test the ADAM architecture using various processor configurations in order to learn more about the dynamical behaviour of the machine. We will investigate several methods for load balancing and will try to improve the performance of the system by removing several bottlenecks. Simultaneously, we will work on the implementation of a compiler for the functional language EVE and experiment with several parallelelization and synchronization strategies. The overall goal of our work is to show the interrelationship between the paradigms controlflow and dataflow and to investigate how one model can be used to simulate the other one.

References

[1] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[2] Kei Hiraki, Toshio Shimada, and Kenji Nishida. A Hardware Design of the SIGMA-1, a Data Flow Computer for Scientific Computations. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 524–531. IEEE Computer Society, August 1984.

[3] Robert Alan Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report MIT/LCS/TR-418, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[4] Gregory Michael Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[5] Wolfgang Schreiner. ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language. Master's thesis, Johannes Kepler University, Linz, Austria, September 1990. Also: RISC-Linz Technical Report 90-42.