The ADAM Abstract Dataflow Machine

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Bitnet: K315640@AEARN

February 13, 1996

Abstract

This paper describes the design and implementation 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 two-address instruction set augmented by primitives for the creation and synchronization of parallel tasks that allow the simulation of dataflow behavior. Special memory units on each ADAM module are responsible for the management of non-strict data structures. The system will be programmed in the non-strict functional (dataflow) programming language EVE that will be compiled to ADAM machine code. 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 implicit parallelism inherent in all algorithms on machines with hundreds or even thousands of processors. Several non-von Neumann architectures have been proposed 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. Especially for **functional and logic programming languages** as well as for applications like **automated theorem proving** that provide a very large amount of implicit parallelism, such kinds of architectures are of great interest.

Dataflow architectures seem to be one of the most promising of these approaches. Since the early 1970s, several dataflow computers have been designed, simulated and partially also built in hardware. Some recent examples for such actually built machines are the *Manchester Dataflow Computer* (Gurd *et al.* 1985), the *ETL SIGMA-1* at the Japanese Electrotechnical Laboratory (Hiraki *et al.* 1984) and the *ETS Explicit Token Store "Monsoon"* at the MIT (Papadopoulos and Culler 1990).

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 that has been 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. Moreover, the dataflow paradigm of program execution requires substantially more complex hardware support than the conventional von-Neumann princple. Therefore, several attempts have been made to combine the benefits of both worlds, of data-driven and of control-driven architectures (Iannucci 1988).

In this paper, the main results of the author's diploma thesis (Schreiner 1990) are presented. This thesis dealt with the design and the implementation of a dataflow simulator on conventional parallel hardware. For this purpose, we designed a hybrid two-level concept combining the advantages of von-Neumann processors with those of the dataflow model. The result of this work is the ADAM Abstract Dataflow Machine that achieves dataflow-like behaviour by basically control-driven program execution. This abstract machine has been successfully implemented by a simulator that interpretes the ADAM machine code on a multi-transputer system. A compiler for the high-level functional programming language EVE is currently under implementation.

However, in spite of this dataflow research we want to emphasize that we are actually not interested in hardware design *per se*. Actually, our work is motivated by the fact that the dataflow style of programming has essential benefits that have no competitive counterpart in parallel programming based on conventional (imperative) languages. We want to show how both worlds, i.e. data-driven and control-driven execution, are related and how the former can also be built on top of the later one.

The following sections describe in a top-down fashion the different levels of the ADAM design and operation principles and also give a short introduction of how to compile a declarative language into ADAM machine code. A substantially shorter description of the ADAM has also been published in (Schreiner 1991a).

2 Basic Idea

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

$$\begin{aligned}&(a+b)*(-y)\ \textbf{where}\\ &\quad a=-x\\ &\quad b=x/y\end{aligned}$$

This expression should be viewed as the body of a function $f(x, y)$ with formal parameters x and y. In the dataflow model, this body is translated into a directed acyclic graph (the *dataflow graph*) as depicted in Figure 1.

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

Figure 1: A Dataflow Graph

```
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 usually translated into explicit `copy` instructions that duplicate their input values. If the successor is a binary instruction, the predecessor must also specify which of the "input ports" of the successor (the left or the right one) is actually referenced.

The execution of a dataflow program proceeds as follows: between different instructions, values are transmitted by pieces of information called *tokens*. These tokens flow along the arcs of the dataflow graph and trigger the execution of the instructions on their way *(data-driven execution)*. If some token passes a binary instruction, i.e. an instruction with two input arcs, a special synchronization mechanisms has to ensure a *rendezvouz* with the partner token: the instruction may only execute and *fire* (i.e. produce a result token) if both input tokens have arrived (the so called *firing rule*).

In the ADAM model, the interpretation of a dataflow graph is substantially different. The arcs of the graph denote registers and the nodes represent two-address instructions operating on these registers:

```
0:   copy   x1, x2, 2
1:   copy   y1, y2, 4
2:   split  3, 6
3:   neg    x1, x1, 8
4:   split  5, 6
5:   neg    y2, y2, 10
6:   wait   e0, 7
7:   div    x2, y1, 8
8:   wait   e1, 9
9:   add    x1, x2, 10
10:  wait   e2, 11
11:  div    x1, y2, 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, i.e. each instruction performs a jump to its successor. Since the above program has two input arguments that may asynchronously arrive, two threads of control enter the code block at addresses 0 and 1, respectively. `split` instructions are used to fork a thread of control into two parallel threads starting at the specified successor instructions. 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 basically the only instructions related to parallelism. However, together with the function call mechanism (discussed in section 6), 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. The compiler may optimize register allocation in such a way that the overall number of required registers is reduced to a minimum.

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** by the use of `wait` statements[1].

Using both optimizations, register sharing and serialization, 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 input threads 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.

After this introduction to the basic principles of our approach, the following sections present the ADAM architecture in more detail.

3 Global Architecture

The ADAM is a parallel machine with distributed memory. It consists of several equally constructed worker modules that are connected by a network of unbuffered uni-directional channels. These channels represent the only communication medium of the ADAM; there is nothing like some global memory that all modules might have access to. 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. On the other hand, the worker modules perform the actual computational tasks.

[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* (Peyton Jones 1987) has to detect to which extenct function bodies may be safely serialized.

Figure 2: The ADAM Network

Figure 2 shows a sample network of ADAM modules (each double-arrow represents two channels for communication in both directions).

Each ADAM worker consists of three components as illustrated in Figure 3:

- **The Arbiter** is the only unit with an interface to the neighbour modules. When the ADAM is booted, the arbiter is responsible to distribute the program to be executed over the network. During program execution, it receives messages from the other local units and from the neighbour modules and routes them to their correct places of destination.
- **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. Moreover, it communicates with its memory unit and those on other modules by request and result messages.
- **The Memory Unit** contains a heap-organized store holding tuples, i.e. data structures of variable 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 references to other tuples) 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

Figure 3: An ADAM Worker Module

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 (either to the local processor or memory unit or to some neighbour module).

4 The Processor Unit

4.1 Construction

The ADAM processor is basically a conventional control-driven processor with several extensions that allow the efficient simulation of dataflow behaviour. The ADAM therefore represents a hybrid two-level design combining features from conventional register architectures and from true dataflow computers.

The processor consists of the following components (depicted in Figure 4):

- **The Execution Unit** is the heart of the processor performing the arithmetical/logical computations. Program execution is basically control-driven but constrained by special *synchronization instructions* that are used to express the data dependences within a program.

- **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 4: 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, the processor continues execution with one thread while it puts a trigger for the other thread into this store
- **The Message Store** and **the Request Store** buffer all messages received. The message 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 described in the following subsection.

4.2 Execution

Within a processor, different function activations (also different activations of the *same* function) are simultaneously alive. Function activations represent heavy-weight processes

Figure 5: Interaction between Processor Components

(tasks) that may be distributed among processors (*function-level parallelism)*. Each activation has therefore its own address space, communication between different activations is only possible via messages. However, also within an activation several instruction sequences may be concurrently executed. These sequences represent light-weight processes (*threads*) that share the same address space and may directly communicate via memory locations *(instruction-level parallelism)*.

The execution the processor now proceeds as follows (see Figure 5): each time the execution of a new function activation is started, the description items of this activation (i.e. the base addresses of the code block and of the register frame) are stored into two *context registers*. Any register number encoded in an instruction is therefore actually an offset to the base address of the frame and every function activation has therefore 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 also its private sets of events each of which is represented by a bit in the frame.

Instruction execution proceeds in a control-driven fashion, i.e. if an activation is started, there is one thread of control executing one instruction after the other. The sequence of instructions is denoted by the extra address field of each instruction. Each instruction operates on the local set of registers in the usual way.

If a `split` instruction is executed, the actual thread of control is split up into two parallel threads. The start address of one of the branches is pushed on the trigger store while the other branch is continued by the current thread. If the thread will be eventually interrupted (e.g. by a `wait` instruction as explained below), then one of the triggers will be popped from the store and used to restart execution at the desired position.

This mechanism therefore manages some kind of *multi-tasking* of parallel threads within function activations.

If a `wait` instruction is executed and fails, the actual thread is terminated. Then, as explained above, a trigger is popped from the trigger store and a new thread is started within the current activation (i.e. the context registers remain unchanged). Only if the trigger store is empty, a context switch has to be made, i.e. a new activation is started and the context registers have to be redefined. The start of a new activation 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. The context in which this thread shall execute is also encoded in the message as a pointer to the context store (delivering the base addresses of the context's activation frame and code block). The message value itself may be either an input argument for the function activation or the result value delivered by some other function 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. When the processor decides to fulfill this request, a new activation frame has to be allocated. Then a free descriptor in the context store is initialized with the base addresses of the activation frame and of the function's code block. Finally, a reference to this 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.

As described above, the ADAM processor represents a hybrid concept combining features of register architectures and of dataflow machines:

- Within a function activation, sequential threads are executing in a **control-driven** fashion: each thread executes one instruction after the other each of which performs some register operation. Concurrent threads within the same activation are synchronized by the application of `wait` statements.

- Between function activations, execution is controlled by messages in a **data-driven** way: each message transmits a value from one activation to another and starts a new thread of control in the target activation. Special request messages are used to create new function activations, possibly on different processors.

The ADAM concept therefore corresponds to a dataflow model where the nodes of the dataflow graph are whole user-defined functions. Inside the nodes, execution proceeds as on conventional von-Neumann architectures.

Figure 6: The ADAM Memory Unit

5 The Memory Unit

5.1 General

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 Figure 6):

- **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. 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`). Furthermore, it is responsible for garbage collection i.e. for reclaiming the heap space that is occupied by tuples that are nowhere referenced any more.
- **The "Expensive" Store** and **the "Cheap" Store** hold all incoming requests classified into two categories: the request may require the allocation of additional heap memory ("expensive") or not ("cheap"). This distinction helps to manage the heap store more efficiently.

5.2 Memory Access

All tuples are *non-strict* i.e. tuple references may be exchanged between parallel processes even if (some or all of) the tuple 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! The one process (the *producer*) may therefore fill the tuple fields with values while the other process (the *consumer*) simultaneously reads these fields and processes their values. This form of **producer-consumer parallelism** is one of the main sources of parallelism in declarative programs ((Arvind *et al.* 1987)).

However, in order to preserve determinism of program execution (i.e. to avoid that the consumer process reads some undefined tuple slot and thus receives a random value), special synchronization tools have been introduced: Each field of a tuple has associated a **full/empty flag** indicating if a value has yet been written into this field or not. These flags are maintained by the heap manager in the following fashion:

- **Allocation Requests:** If a request arrives to allocate space for a new tuple, all fields of this tuple are marked as empty and a reference to this tuple is returned to the sender of the request.
- **Read Requests:** If a read request arrives for a field that already contains some value, the value of this field is returned to the sender of the request. However, if the field that has not yet been defined, the request is put into a waiting queue and a reference to this queue is stored in the field.
- **Write Requests:** If a write request for some tuple field arrives, all waiting read requests queued into that field are released: the value delivered by the write message is returned to the senders of the deferred read requests and is then written into the field (setting the flag to full). However, if an attempt is made to overwrite an already defined tuple field, an error is reported: all tuples have to obey the *single-assignment rule* which is essential for receiving deterministic result values in parallel systems.

Access to the services provided by the memory unit is entirely handled via the use of these messages. However, an ADAM program has a more abstract view of the memory access which is described in the next subsection.

5.3 Memory Instructions

Corresponding to these tree types of memory requests, there exist three instructions `alloc`, `read` and `write` in the ADAM instruction set. From the point of view of program execution, all three instructions are non-blocking: `alloc` and `write` immediately return their results, i.e. a reference to a new tuple and the contents of some tuple field, respectively; `write` immediately writes the desired value and then continues execution.

However, `alloc` and `read` are actually **split-phase operations**: Having submitted the memory requests into the network, the processor immediately interrupts the execution of the actual thread and starts another one. The interrupted thread is only continued at the arrival of the desired value. In case of the `write` instruction, execution is continued immediately after the write request has been submitted (i.e. *before* the actual write operation has actually been performed).

This behavior of the memory access instructions has a crucial consequence on the overall performance of the system: since the actual process does *not* wait for the result of a memory operation, the architecture is able to **tolerate extremely long memory latencies**. It does not matter how long the message and the delivered result is on its way, as long as there are other processes still to be executed. 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.

5.4 Garbage Collection

The ADAM architecture uses for garbage collection an improved version of the **weighted reference counting algorithm** (Bevan 1987). This garbage collection method has (compared to all other garbage collecting schemes) the advantage of being very simple and easily adaptable for distributed memory architectures: 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. Morover, it is also rather memory efficient only requiring an overhead of a few bits per reference.

The basic idea of this method is to associate to each reference a *weight* and to each tuple a *reference count*. The algorithm then always maintains the invariant

> **The reference count of a tuple is equal to the sum of the weights of all references to it.**

This ensures that, if a tuple is not referenced any more, its reference count drops to zero which signals that the tuple space may be reclaimed.

In detail, this algorithm works as follows (see Figure 7):

- **Allocating Tuples:** If a new tuple is allocated, the tuple is given some reference count that is a power of two. The binary logarithm of this count is stored as the weight in the new reference that is returned as the result of the allocation request.

- **Deleting References:** If a reference is not used any more, it is deleted by sending a *decrement message* to the memory unit that holds the referenced tuple. The reference count of the tuple is then decreased by the weight of the deleted reference (taken to the power of two). If the count drops to zero, the tuple may be reclaimed.

- **Duplicating References:** If a reference is duplicated, both of the copies get half of the weight of the original reference. Since weights are stored as their binary logarithms, this means to decrease the weight field by one.

  However, if the weight would drop to zero, the reference must not immediately be copied: first, an *increment request* has to be sent to the memory unit that contains

Figure 7: The Garbage Collection Mechanism

the referenced tuple. The reference count of this tuple is increased by the maximum weight minus one (considering the weight of the old reference) and a new reference is returned with a weight set to its maximum. This reference may now be duplicated as usual.

If the reference count of a tuple already holds the maximum value to be represented, this value is considered as "infinity". All further increment and decrement requests will not have any effect on the reference count any more and the tuple will never be reclaimed.

The use of this weighted reference scheme has the additional advantage that communication between processors 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. 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 then neglectible.

6 The Function Call Mechanism

A general description of how to compile a high-level declarative (functional or logic) programming language into code for the ADAM architecture definitely goes beyond the scope of this paper. The interested reader may refer to the still ongoing diploma thesis (Loidl 1992).

However, for understanding the operation principles and the benefits of the ADAM architecture, it is important to know about the function call mechanism that serves as the interface betweeen data-driven execution (between different function activations) and control-driven execution (within an activation). Generally, the compilation of a function call requires the generation of ADAM code for the following tasks:

1. **Activation:** a request for a new function activaton has to be submitted;
2. **Arguments:** the function arguments have to be sent to the new activation;
3. **Result:** the result value of the activation has to be received.

For example, a call $f(a, b)$ of a binary function f with arguments a and b will be compiled into some piece of code that is similar to the following one:

```
i:      call   f
i+1:    recv   r, m
i+2:    recv   s, i+3
i+3:    split  j, k

j:      wait   e0, j+1
j+1:    send   a, r3, 0
j+2:    jump   k+2

k:      wait   e1, k+1
k+1:    send   b, r3, 1
k+2:    wait   e2, n
```

There are **three concurrent threads of control** that enter the code at addresses i, j and k, respectively:

- **The 'f' thread:** The first thread enters at address i as soon as it becomes certain that the result of f is required and hence a new function activation has to be created. By executing the `call` instruction at address i, a request message is submitted to some processor in the network that will eventually start a new activation of f. However, the actual thread is interrupted immediately after having submitted the request.
- **The 'a' thread:** The next thread enters at address j as soon as the value of the first argument of f has become available in register a. However, the thread immediately encounters a `wait` e_0 statement and is therefore blocked.
- **The 'b' thread:** Similarly, the last thread enters at address k as soon as the value of the second argument of f has been computed into register b. The synchronization statement `wait` e_1 will block this thread, too.

The request message sent by the first thread will cause some processor in the network to create a new activation of f and start a thread in this activation (see the explanation below). However, the processor will immediately return the descriptor of this new f activation to the caller. The caller receives this descriptor by the `recv` statement at address $i + 2$ into register s and starts a new thread there.

This thread is split at address $i+3$ into two parallel threads starting at addresses j and k. At these points, the threads are synchronized by `wait` instructions with the corresponding threads that have computed the argument values a and b, respectively. Each argument value is independently sent by a `send` instruction to the new f activation. Finally, both threads synchronize themselves by the `wait` at address $k + 2$.

Eventually, f will compute its result value and return it to the caller. The caller receives the value by the `recv` instruction at address $i+1$ and continues execution by a new thread starting at address m.

The whole code block above is exited by **two concurrent threads of control**:

- **The result thread:** This thread starting at address m signals that the result of the function call has been delivered into register r. Any successor instruction in this thread can rely on this.

- **The argument thread:** The second thread at address n signals that both argument values a and b have been submitted. This thread is necessary, because it is possible that a function returns a proper result even if not all arguments have yet been submitted to that function (see the explanation below).

While the first exit thread delivers the result of the function activation, the second one is necessary in order to detect when all outstanding arguments have been received and all instructions have properly executed. Only then the actual function activation may be terminated.

The reader may wonder, why in this function call mechanism separate concurrent threads are used to activate a function and to submit its arguments. The reason for this is the **non-strict semantics** of the language that we have in mind for programming the ADAM architecture. Consider e.g. the function definition

$$f(a,b) = \begin{array}{l} 0,\ \textbf{if}\ a > 0 \\ b,\ \textbf{else} \end{array}$$

On a function call $f(1, b)$, we need not know the value of b in order to determine the result of f, i.e. f is *non-strict* in its second argument. Operationally spoken, the execution of f may start and perform some sensible operations (namely the test $a > 0$) while the value of b is still under computation. Generally, during the evaluation of expressions of the form $f(g(x))$, the execution of f may go on *in parallel* with the execution of g.

This form of *vertical parallelism* (i.e. the parallel execution of nested function calls) is complementary to the better known *horizontal parallelism* (i.e. the parallel evaluation of function arguments) but can be only exploited in languages with non-strict semantics: in such languages, the execution of a function must *not* wait for the values of its arguments but proceed in an independent fashion.

The above piece of code just implements the non-strict semantics of a function call with the activation of a function being handled by a separate thread that runs in parallel with the threads that are in charge of the parallel argument evaluation. However, optimizations may detect cases in which this general form of function call is superfluous and then synchronize the function thread with the argument threads *before* the function body starts to execute.

7 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

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 8: Several ADAM Benchmarks

(Schreiner 1990) on conventional parallel hardware using principles of dataflow architectures. While the EVE compiler is still under implementation, the implementation of the ADAM concept has already been finished:

The target hardware consists of a distributed memory multiprocessor containing 16 INMOS T800 transputers that may be dynamically configured to the desired topology. On this system, a simulator for the ADAM concept has been implemented in the OCCAM2 language. This simulator is organized in such a way that each transputer simulates one ADAM module where the components of the module (arbiter, processor, memory unit and their subcomponents) are implemented as processes concurrently executing on the same physical processor.

For debugging and testing purposes, a PC-based ADAM assembler has been implemented generating ADAM machine programs from manually written ADAM assembly programs. In the final system, the compiler will generate this assembly code from programs written in the EVE language.

Using the assembler, several test programs have been hand-coded and show rather promising speedups. Some of the results are listed in Figure 8. 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 lots of intermediate lists) and PRIMES1 (a prime number generator making use of the concept of difference lists). The efficiency (speedup/processor ratio) ranges in both of the first test programs from more than 90 % to about 45 %. From these results, 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 propbably a consequence of the load balancing which is not yet satisfactory.

8 Future Work

Our future work on this project will concentrate on three aspects:

1. **The Eve Compiler** is currently under implementation and will be finished until the summer of 1991. When the first prototype is available, we will be able to write bigger programs and learn more about the dynamical behaviour of the Adam architecture.

2. **Symbolic Algorithms:** One of the main goals of our work is to detect the suitability of the dataflow concept for purposes of symbolic computation (computer algebra, automated theorem proving, computational geometry, ...). We will therefore implement some symbolic algorithms in use at our institute in the Eve language and learn about the behaviour of this class of applications on our system.

3. **Optimizations:** From the results of these experiments, we will probably be able to optimize and maybe partially redesign the Adam concept in order to avoid several possible bottlenecks. Moreover, we will experiment with different code generation strategies in the compiler to detect the actual sources of parallelism in declarative programs.

Moreover, the author currently investigates a possible topic for his Ph.D. thesis to which the findings of the Adam concept might lead (Schreiner 1991b): The idea is to restrict the extensive Adam runtime system that supports the main features of dataflow architectures and interpretes an abstract machine code to a minimal set of high-level functions that can be added to any sequential language (e.g. C). A compiler could then translate a declarative language to this explicitly parallel imperative form.

The result would be a highly portable system consisting of a compiler generating standard C code and a few machine-dependent functions that have to be rewritten for each parallel back-end. In this sense, the dataflow style of programming could be also efficiently supported on conventional parallel architectures which would certainly help to make its features and advantages more widely known and used.

References

[Arvind *et al.* 1987] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987. Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.

[Bevan 1987] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

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

[Hiraki *et al.* 1984] 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.

[Iannucci 1988] 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.

[Loidl 1992] Hans Wolfgang Loidl. Compiling Functional Programs to Dataflow Code. In *Proceedings of the Workshop on Parallel Computers and Programming Languages*, number 10 in Communications of the German Computer Society, Group for Parallel Algorithms, Computer Architectures and System Software, pages 78–86, Dagstuhl Castle, Germany, February 26–28, 1992. Also: Technical Report ACPC/TR 92-10, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[Papadopoulos and Culler 1990] Gregory M. Papadopoulos and David E. Culler. Monsoon: An Explicit Token-Store Architecture. In *The 17th International Symposium on Computer Architecture*, number 18(2) in ACM SIGARCH Computer Architecture News, pages 82–91, Seattle, Washington, May 28–31, June 1990.

[Peyton Jones 1987] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, New York, 1987.

[Schreiner 1990] Wolfgang Schreiner. ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language. Master's thesis, Johannes Kepler University, Linz, Austria, October 1990. Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.

[Schreiner 1991a] Wolfgang Schreiner. ADAM — An Abstract Dataflow Machine and Its Transputer Implementation. In *Distributed Memory Computing, 2nd European Conference, EDMCC2*, pages 392–401, Munich, Germany, April 22–24, 1991. Volume 487 of Lecture Notes in Computer Science, Springer, Berlin. Also: Technical Report 90-53, RISC-Linz, Johannes Kepler University, Linz, Austria, November 1990. Also: Technical Report 91-3, Austrian Center for Parallel Computation, January 1991.

[Schreiner 1991b] Wolfgang Schreiner. Macro-Dataflow in C. In *2nd Scientific Meeting of the Austrian Center for Parallel Computation (ACPC)*, Wilhelminenberg Castle, Vienna, Austria, March 8–9, 1991.