A first investigation of the crossflow computer.

Acknowledgement. This text was written while drs R.W.Bulterman, ir.W.H.J. Feijen, ir.A.J.Martin and drs.F.J.Peeters were looking over my shoulder. It summarizes discussions in which dr.M.Rem participated as well.

The purpose of this note is to investigate an alternative to the architecture of (instruction) pipe-lining. It has been inspired by the following objections against pipe-lining.

1. The interlocking of pipe-lined machines is very extensive and complicated. This has several undesirable consequences.

1.1. It is very expensive to design.

1.2. It is very difficult to prove the correctness of the design.

1.3. It is expensive to make and hard to maintain.

1.4. It violates by its very structure one of the basic assumptions of traditional sequential hardware, the assumption being that if each instruction, all by itself, is verified to operate correctly, a sequence of instructions will be executed correctly as well; this assumption is justified in traditional sequential hardware because there successive instructions can only interfere via explicitly named, discrete memory elements --storage cells and registers-- . "Checking" the correct operation of a pipe-lined machine with its intertwined instruction executions seems orders of magnitude more difficult, if not impossible.

2. The use of a pipe-lined machine presents conflicts because at each conditional jump it is hard to decide how to direct the instruction fetch. Either we stop --but then the effect of pipe-lining is temporarily paralyzed-- or we guess.

2.1. Guessing is unattractive. First of all it requires the whole machinery for "undoing" in the case of a wrong guess; secondly it exerts strong pressures upon the programmer --or the compiler?-- to make on too detailed a level the program text dependent on expected frequencies.

2.2. I have been told that designing an optimizing compiler for a pipe-lined machine is fraught with unpleasant surprises.

2.3. I have been told that effectuating an interrupt in a pipe-lined machine in an orderly fashion has, in many cases, been too difficult for the designers.

We intend to explore the so-called "crossflow computer" as an alternative

machine architecture that may achieve the speeding-up effects of pipe-lining, but hopefully without most of the complications listed above. Stated very shortly: we would like to harvest the fruits of concurrent processing without paying the price of extensive synchronization protocols, and without introducing the extensive, diversified hardware. More precisely: we envisage a machine consisting of a modest number--six, say-- in hardware identical components, each component in principle not much more complicated than a traditional sequential machine. In order to stress the fact that these components should be viewed as executing together --and in rather strict synchronism-- the same algorithm, we shall consistently refer to them as "components": the components together constitute a single sequential machine.

In order to avoid false expectation we would like to stress, right at the beginning, that as far as the crossflow computer can be regarded as an exercise in concurrency, the exercise is very modest. We will be perfectly happy if we arrive at an architecture which, by means of six times as much hardware, reduces execution times by a factor two or three. We are perfectly aware of the fact that our approach is not applicable if someone wants to reduce computation time by a factor of ten at the possible expense of twenty or fourty times as much equipment.

As a matter of fact, preliminary investigations have indicated --and, as I have heard, this is confirmed by experience elsewhere-- that we must be glad if we manage to keep our, say, six components happily productive, when our start is a traditional sequential algorithm. This upper bound has an interestin possibility of which we have been aware right from the start. (At the moment of writing it has not been explored yet.) The possibility opens up when the machine is equipped with a few more components: if they cannot be used to speed up the computation, they may be available for run-time checking!

Note. Run-time checks are, by definition, extremely skew and, as such, yield very little information. If run-time checks are performed at the expense of progress speed, there is, therefore, a strong pressure to omit them during production runs --with all undesirable consequences-- . In practice, the dilemma has always been solved by building-in dedicated, concurrently active equipment, the prime examples being the parity check and the interrupt detection. A few

additional components could perhaps be dedicated to program dependent run-time checking, that could then be performed without noticeably slowing down the computation proper. (End of note.)

* * *

Let us try to give and motivate a --by necessity in this stage rough-- description of the kind of machine we intend to investigate. The machine will consist of a modest number of identical "components". Each component will comprise a private high-speed memory and a purely sequential processor. Information exchange between the components is assumed to take place via a common bus, with the fundamental assumption that during a time slot in which the bus is used, information can be broadcast by <u>one</u> of the components and be picked up by <u>all</u> the others.

<u>Note</u>. For the time being we shall assume that the bus is very fast compared with the processing. The extent to which this assumption is essential or a matter of temporary convenience --e.g. for the ease of compiling-- remains to be seen. (End of note.)

<u>Note</u>. The bus is assumed to be a galvanic contact between the components, with all the receivers "reading" simultaneously. The number of components is assumed to be small enough so that fan-out presents no serious problems. (End of note.)

<u>Note</u>. The assumed speed of the bus and "clock skew" could present problems; for the time being we hope that they can be solved. (End of note.)

Each component looks like a sequential machine operating under control of the program as stored in its own store. Just as we introduced the term "component" versus the whole "machine", we introduce, in order to avoid confusion, the term "fragment" versus the whole "program". When the whole machine executes a program, each component executes its fragment. We consider it the task of the compiler to construct from the source program the, say, six fragments that will control the activity of the, say, six components.

<u>Note</u>. The division of the total workload over the six components is decided

statically (if you know what I mean). We consider the six components to form one machine: when one of the components is out of order, the machine is down. The so-called "graceful degradation" is most definitely not our target. Everybody who wants that, should couple a number of crossflow machines. (End of note.)

The six components are assumed to work in a rather strict synchronism. I.e. not only will they be controlled by a common clock, but even the execution of the six fragments is synchronized to the extent that we could, as a first approximation, visualize the six fragment executions taking place under control of a central ("virtual"?) instruction counter. In practice we don't expect this to be truly the case. Firstly, instruction counters locate instructions in the stores and dynamically corresponding fragment parts in different components could require different amounts of store. Secondly, it is not excluded that certain redirections of control --possibly, for instance, the return from a procedure-- don't take place simultaneously in all six components: some of them might be mildly our of phase. It should be pointed out that we expect the pattern of flow of control of the program fairly faithfully reflected in each of the fragments: a loop on the program is practically certain to give rise to a corresponding loop in each of the fragments.

A point of special concern has been the problem of bus allocation; the bus is clearly a common resourse and it seemed mandatory that the bus can be used without any form of handshaking, either for the purpose of synchronization or for the purpose of conflict resolution between two or more senders. We have therefore decided that the possible patterns of bus usage are fixed during compilation (or rather "fragmentation") and are recorded in the fragments to such an extent that for each component its fragment prescribes when to send or to receive what. This decision makes quite clear the extent to which the various fragments derived from a single program have to be derived simultaneously and in strong interdependence of each other: at run time the sending of a result by one component under control of its fragment must coincide with the reading from the bus by one or more other components under control of their fragments. There will be no contention, i.e. if during a time slot more than one component sends, there must be something wrong --either component malfunctioning or an erroneous compiler-- . A hardware check on such erroneous bus usage is assumed.

For the synchronization of the bus contacts various techniques can be considered. Under the assumptions that

a) each instruction takes a fixed, a priori known number of time units

b) the order code contains the instructions "SEND" and "RECEIVE" , that transmit information from a component register to the bus and vice versa respectively

c) the order code contains DELAY instructions, each of a fixed, a priori known number of time units,

the fragments can be coded in such a way that during concurrent execution --as if by magic!-- each SEND coincides in time with the appropriate RECEIVE's. The obligation of "padding out" the fragments is tedious, but not difficult. Alternatively we could consider each RECEIVE to extend up to the end of the "next" SEND. Note that also in this case the fragmentation cannot be performed without rather detailed timing information: we would have to ensure that no SEND starts to early. With the above we don't think that we have explored all possibilities: at this stage of our investigations we would prefer not to commit ourselves beyon the decision that during fragmentation the pattern of bus usage is determined.

Note. We have not fully committed ourselves yet on what the pattern of bus usage may depend. It will certainly depend on the flow of control --which, therefore, is largely common to all fragments--, it could in principle also depend on intermediate values: if during a specific time slot a number of components are going to receive the value of $a[i]$, stored in another component, the identity of the sender could be made dependent on the value of i , viz. when the elements of the array are distributed over more than one component. (This is not what we emphatically envisage; on the other hand this is not the time to exclude it.)

A second example may be provided by an implementation of (in old notation)

if B then x:= y else x:= z fi ;

if x , y , and z are stored in different components, the one containing x may receive its new value in a sense from "an unknown sender". Both the examples illustrate how (local) differences in flow of control may be confined to part of the fragments. (End of note.)

Note. It might be thought that timing obligations are hard to fulfill because execution time is not known a priori in the case of repetition. In that case, however, the repetition will be faithfully reflected in all fragments. (End of note.)

Note. If we allow nondeterminacy in our source program, e.g.

$$\underline{\text{if}}\ B1 \rightarrow S1 \ [\!]\ B2 \rightarrow S2\ \underline{\text{fi}}$$

the nondeterminacy has to be resolved on the level of the total machine, and not independently in the different components. (End of note.)

Note. In sequential programming we have already experienced the luxury of programming as if all guards of a guarded command set are evaluated concurrently. If the values of the different guards are formed in different components and they have, therefore, to be broadcast, the bus, which is assumed to have many wires, may be split up in a number of sub-buses. (End of note.)

* * *

In order to describe the kind of happenings inside the working crossflow computer as we visualize it at the moment, we relate them to what would happen during a straightforward sequential implementation of the same program. (The fact that that relation is such a close one is indicative for the modest concurrency in our proposal.) We envisage that to each transmission via the bus will correspond either a guard-broadcasting or an assignment from the traditional sequential execution. Conversely we hope that many store instructions in the traditional sequential execution --in particular those to anonymous intermediate results-- won't need the bus. It is envisaged that the bus contacts will take place in exactly the same order as in which the corresponding events would take place in the sequential execution, a correspondence to which we wish to stick for two reasons. The one reason is that without more revolutionary massaging of the program text, the design of the compiler/fragmenter is already difficult enough; the other reason is that we wish our reasoning techniques about sequential programs to remain applicable.

(To be continued.)

Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow