F. Preparata (Ed.)

CIME Summer Schools

Theoretical Computer Science

68

Bressanone, Italy 1975

FONDAZIONE CIME ROBERTO CONTI

F. Preparata (Ed.)

Theoretical Computer Science

Lectures given at a Summer School of the
Centro Internazionale Matematico Estivo (C.I.M.E.),
held in Bressanone (Bolzano), Italy,
June 9-17, 1975

C.I.M.E. Foundation
c/o Dipartimento di Matematica “U. Dini”
Viale Morgagni n. 67/a
50134 Firenze
Italy
cime@math.unifi.it

ISBN 978-3-642-11118-1 e-ISBN: 978-3-642-11120-4
DOI:10.1007/978-3-642-11120-4
Springer Heidelberg Dordrecht London New York

©Springer-Verlag Berlin Heidelberg 2011
Reprint of the 1st ed. C.I.M.E., Ed. Cremonese, Roma 1975
With kind permission of C.I.M.E.

Printed on acid-free paper

Springer.com

CENTRO INTERNAZIONALE MATEMATICO ESTIVO

(C.I.M.E.)

I Ciclo - Bressanone dal 9 al 17 giugno 1975

THEORETICAL COMPUTER SCIENZE

Coordinatore: Prof. F. PREPARATA

R. E. MILLER	: Parallel program schemata	pag.	5
D. E. MULLER	: Theory of automata	»	65
R. M. KARP	: Computational complexity of combinatorial and graph-theoretic problems	»	97

CENTRO INTERNAZIONALE MATEMATICO ESTIVO

(C.I.M.E.)

PARALLEL PROGRAM SCHEMATA

R.E. MILLER

Corso tenuto a Bressanone dal 9 al 17 giugno 1975

Configurable Computers and the Data Flow Model Transformation

R. E. Miller

In this lecture we discuss a type of computer organization which is based upon the concept of operation sequencing being controlled by operand availability. Such sequencing differs radically from current computer organizations in which the sequencing is determined by an explicit or implicit ordering of the instructions which is controlled through an instruction counter which specifies the next instruction to be performed. We will discuss a particular type of data sequenced computer called "configurable computers" [7]. Closely related types of machines have subsequently also been proposed and studied by others [3, 11, 10, 9].

One of the major problems with unconventional computer organizations that have been proposed in the past is the great difficulty of programming such machines. In some cases the proposed structures were so complex that an inordinate amount of work was required to specify what each object in the structure was to do, while in other cases the structures were particularly suited to only a narrow class of computations, and were ill suited for general purpose use. A number of approaches for representing the computational process in a data sequenced manner have recently been proposed [11, 5, 2, 1]. We will briefly describe one such approach [8, 6] which is aimed at automatically transforming normal computer programs into a data sequenced form suitable for configurable computers. Even though a direct representation of an algorithm into a data sequenced form may be able to better exploit the advantages of the data sequenced form, considerable advantage is gained in automatic transformation -- closely related to optimizing compiler techniques -- which allows the user to continue to use his well known programming languages directly, rather than learning a new complex language immediately.

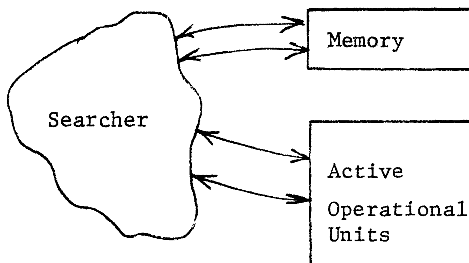
Configurable Computers

One of the central concepts of configurable computers is to have a machine that changes its structure into the natural structure of the algorithm being performed, thereby allowing parallel computations and many of the speed advantages of special purpose devices.

Two particular approaches for attaining this goal are described. The one approach is called Search Mode and the other is called Interconnection Mode. These should be viewed as only two ends of a spectrum of possibilities in which considerable differences in the possible computer control structures exist.

Search Mode Configurables

The basic organization of a search mode machine is depicted in Figure 1.



The operational units are thought of as a set of either general or special purpose units which perform the computational, conditional, and data generation aspects of the computation. When one of these units has completed a task it requests the searcher to find another task for it to perform. The searcher, which is essentially a new kind of control, then inspects memory, or a suitable portion thereof, for a new task for the operational unit. This organization can thus be adopted to the idea that a task becomes capable of being performed when its operands have

been computed. An example of machine instruction format aids in seeing in more detail how the machine could operate. We give an example for an arithmetic operation but, of course, a complete repetoire of instructions and formats would have to be given for a complete specification.

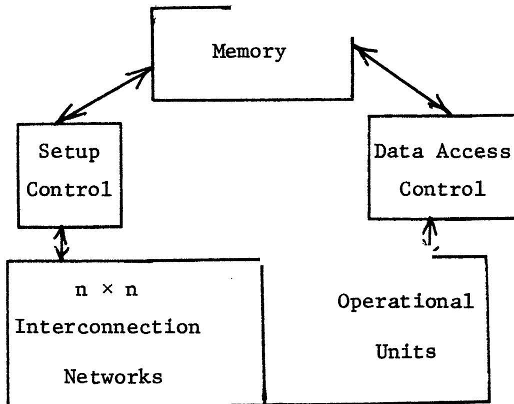
Operation Code	Status Bits	First Operand	Second Operand	"Address" for Result

This instruction format contains first a field of bits to specify the type of operation to be performed (the operation code). Skipping over the status bits for a moment, it then contains fields for the operand values to be stored. Within the status bits one keeps track of whether the operands currently reside in these locations (i.e., have been computed and stored there) and whether the location for the result is available for storing a result. Finally, the "address for result" field specifies where the result is to be stored (i.e., as an operand for some other instruction) Storing a result in an operand field updates the status bits, as does the action of performing an instruction. Clearly, this type of format eliminates the need for normal instruction sequencing. The sequencing is essentially "data driven."

It is the operation code and status bit fields that the searcher inspects to determine readiness of an instruction. This could be imagined to be done by a type of associative search, or by a method of building up stacks or queues of instructions that are ready to be performed. Before describing how programs can be transformed into such a data sequenced form, we describe the interconnection mode machine idea.

Interconnection Mode Configurables

In the search mode concept the sequencing of instructions was done through the action of operation results becoming operands of new instructions, and these transfers of information were done by storing results in the proper places of instructions. With the advent of economical electronic switches more direct connection of result to operand could be envisioned. This is the case for the interconnection mode idea. A block diagram is shown below.



Here the interconnection network is used to directly interconnect outputs of one operational unit to inputs of another operational unit in accord with the result to operand specifications of the algorithm. The basic steps for such a machine are:

1. Decompose the program into appropriate size blocks.
2. Transform each block into a data sequenced form.
3. Store the blocks, so transformed, in memory as setup instructions for the interconnection network.
4. Choose a block to be performed (to start with this is the block with the start of the program -- subsequently this is specified

by what block the running block exits to next) and set up the interconnections as specified.

5. Perform the block execution. Note that during this time no instuctions -- only data -- need be stored or fetched from memory.
6. Termination of block specifies next block to be performed (return to 4).

At this point it should be clear that both the search mode and interconnection mode machines are sequenced essentially by data paths rather than control paths through the algorithm. Thus, the natural parallelism of the algorithm can be used to speed up operation. Also, the machines have some of the speed advantages of special purpose devices. This is especially true of the interconnection mode machine since units are actually directly interconnected as they would be in a special purpose device The operational control -- distributed throughout the machine by data availability considerations -- is also rather simple, and this could be a distinct advantage over other approaches to high performance where very complex control mechanisms are required. Finally, as we have indicated, standard programming languages can be used to express the jobs to be done and transformational techniques to provide suitable machine language instructions for these machines should be readily developed using techniques known for compiler optimization. Other potential advantages for these machines are given in [7].

Data Sequenced Program Transformation

We now outline how a program can be transformed into a form suitable for our configurable computers. We call this a <u>data flow transformation</u>. The basic steps of the transformation are:

1. Partition the program into "basic blocks" and name each block. A <u>basic block</u> is a contiguous segment of code which can be entered only through the first instruction of the block, must be executed by executing each successive instruction in order, and can be exited only from the last instruction in the block. I.e., it is a "straight line" segment of code.
2. Determine the immediate predecessors and successors of each basic block.
3. Generate (in arbitrary order) a "data flow segment" for each block. A <u>data flow segment</u> consists of:
   (i) input list -- i.e. variables needed by the block.
   (ii) output list -- i.e. result names at end of block execution.
   (iii) interconnected modules -- i.e. the operations and flow of data from result to operand between operations in the block.
4. Interconnect data flow segments. This uses the predecessor and successor information and updates input and output lists for data "passing through" a block.

Without going into great detail we illustrate the transformation through an example from [6].

<u>An Example of Data Flow Model Transformation</u>

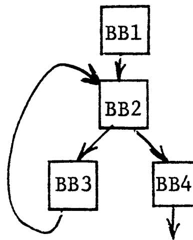
As an example program we consider the problem of evaluating the function $f(x) = a^x + bx + c$. We assume that x, a, b, and c are inputs stored in the symbolic locations x, a, b, and c respectively, and also assume that x is a positive integer. A simple program to perform this evaluation is shown below. The program language used is simple and should be self-explanatory.

Statement #	Program	Comments
1	CLA x	set accumulator to x.
2	STO COUNT	put x in location COUNT.
3	CLA a	put a in accumulator.
4	TRA 6	transfer to statement 6.
5	MPY a	multiply accumulator by a.
6	Decrement COUNT	decrease COUNT by 1.
7	Branch on COUNT (to 5 on ≠ 0)	conditional transfer.
8	STO T	store a^x in T.
9	CLA x	place x in accumulator.
10	MPY b	form bx in accumulator.
11	ADD T	form a^x+bx in accumulator.
12	ADD C	form a^x+bx+c in accumulator.
13	STO R	store result in R.

Applying the notion of basic block to this program we find that instructions 1, 2, 3, 4 form a basic block with instruction 1 being the start of the program. Similarly instructions 6, 7 form a basic block, 8, 9, 10, 11, 12, 13 form a basic block and 5 alone forms a basic block. These blocks are depicted and named BB1 through BB4 in the following diagram.

BB1	1	CLA x
	2	STO COUNT
	3	CLA a
	4	TRA 6
BB3	5	MPY a
BB2	6	Decrement COUNT
	7	Branch on COUNT (to 5 on ≠)
BB4	8	STO T
	9	CLA x
	10	MPY b
	11	ADD T
	12	ADD c
	13	STO R

Step 2 of the algorithm determines immediate successors and immediate predecessors as shown in the following figure.



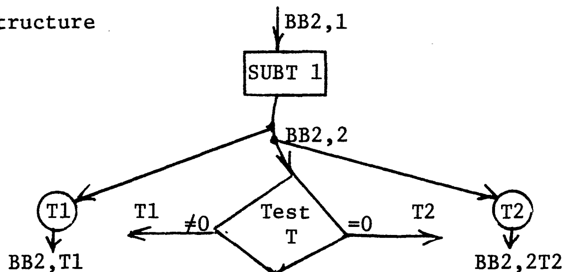
Step 3 of the algorithm generates a "data flow segment" for each basic block. The idea here is to generate a list of items needed as inputs to the block, the outputs created by the block and the operations used to create these outputs along with the flow of data between the operations within the block. The items have names associated with them through the program language definition, these names we call "source names." During generation we assign "local data names" to items also. Consider, for

example, basic block 2, (BB2). This block starts with instruction 6 -- Decrement COUNT. By definition this instruction needs an input with source name "COUNT" and produces a new output also called "COUNT" which has a value less than the original value of COUNT. Thus COUNT is placed on the input list, and we associate a local data name with this input value. We use the name BB2,1; i.e., the first local data name in BB2. In the output list we then have COUNT with a new value and name this new value BB2,2 as a local data name. The operation performed is a SUBTRACT 1 so this operation gets placed in the module structure with input BB2,1 and output BB2,2. The second instruction of BB2 is Branch on COUNT. This instruction uses the current value of COUNT, namely BB2,2 and tests it for =0 or ≠0. This is indicated in the output list as changing item COUNT-BB2,2 to two values COUNT BB2,2T1 and COUNT BB2,2T2 for the outcome of the test either being outcome T1 or T2. A test module is added to the module structure -- we call it test T -- with the two indicated outputs. This completes Step 3 for BB2. Our result is summerized below.

BB2 Data Flow Segment

Input List	COUNT - BB2,1
Output List	~~COUNT - BB2,2~~
	COUNT - BB2,2T1
	COUNT - BB2,2T2

Module Structure



Similar calculations are done for each of the other basic blocks producing the following results.

BB1 Data Flow Segment

Input List
x - BB1,1
a - BB1,2

Output List
~~ACCUM = BB1,1~~
COUNT - BB1,1
ACCUM - BB1,2

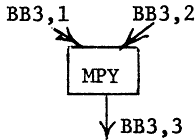
no modules.

BB3 Data Flow Segment

Input List
ACCUM - BB3,1
a - BB3,2

Output List
ACCUM - BB3,3

Module Structure

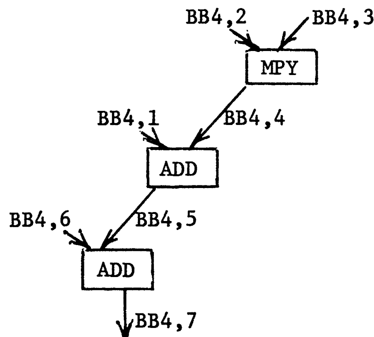


BB4 Data Flow Segment

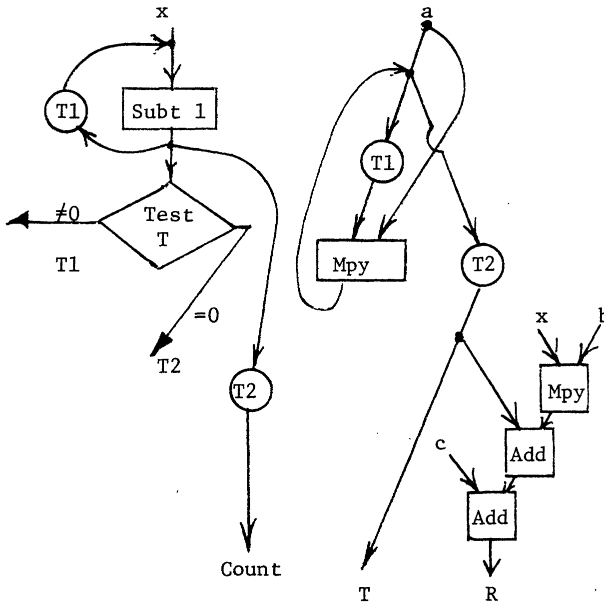
Input List
ACCUM - BB4,1
x - BB4,2
b - BB4,3
c - BB4,6

Output List
T - BB4,1
~~ACCUM = BB4,2~~
~~ACCUM = BB4,4~~
~~ACCUM = BB4,5~~
ACCUM - BB4,7
R - BB4,7

Module Structure



Step 4 of the transformation interconnects these module structures by using successor and predecessor information and making output to input connections through common source names. The result of making these interconnections and inserting test points T1 and T2 for places where data passes only conditionally on the outcome of test T is shown in the next figure. Note that even in this simple example some possibilities for parallelism are exhibited. For example, the two multiplications and the subtract 1 operations could all be performed concurrently.



It should be clear from the descriptions given for configurable computers that this diagram provides all the essential information needed to specify the instructions for a search mode machine or the interconnections for an interconnection mode machine. Clearly, the sequencing so specified differs considerably from how the original program would have run on a conventional computer.

REFERENCES

[1] A. Bährs, "Operation Patterns," Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972, in Lecture Notes in Computer Science, Vol. 5 International Symposium on Theoretical Programming, Springer-Verlag, New York, 1974, pp. 217-246.

[2] J. B. Dennis, J. B. Fosseen, and J. P. Linderman, "Data Flow Schemas," Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972, in Lecture Notes in Computer Science, Vol. 5 International Symposium on Theoretical Programming, Springer-Verlag, New York, 1974, pp. 187-216.

[3] J. B. Dennis and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings ACM Annual Conference, November 1974, pp. 402-409.

[4] K. B. Irani and C. R. Sonnenburg, "Exploitation of Implicit Parallelism in Arithmetic Expressions for an Asynchronous Environment," Report

[5] P. R. Kosinski, "A Data Flow Programming Language," IBM T. J. Watson Research Center Report RC-4264, Yorktown Heights, N. Y., March 1973.

[6] R. E. Miller, "The Data Flow Model Transformation," to appear in the Proceedings of the 1972 GMD Summer Seminar on Systems Organization, Bonn, Germany.

[7] R. E. Miller and J. Cocke, "Configurable Computers: A New Class of General Purpose Machines," Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972, in Lecture Notes in Computer Science, Vol. 5 International Symposium on Theoretical Programming, Springer-Verlag, New York, 1974, pp. 285-298.

[8] R. E. Miller and J. D. Rutledge, "Generating a Data Flow Model of a Program," IBM Technical Disclosure Bulletin, Vol. 8, No. 11, April 1966, pp. 1550-1553.

[9] S. S. Reddi and E. A. Feustel, "A Restructurable Computer System," Report, Laboratory for Computer Science and Engineering, Rice Univ., Houston, Texas, March 1975.

[10] C. R. Sonnenburg, "A Configurable Parallel Computing System," Ph.D. Dissertation, University of Michigan, Ann Arbor, October 1974.

[11] J. C. Syre, "From the Single Assignment Software Concept to a New Class of Multiprocessor Architectures," Report, 1975 Department d'Informatique, C.E.R.T. BP4025, 31055 Toulouse Cedex, France.

Computation Graphs and Petri Nets

R. E. Miller

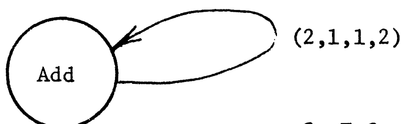
In this lecture we show how a special type of Petri net, widely studied in the literature, can be represented by computation graphs. We then illustrate how results about computation graphs can be translated into results for such Petri nets. We first introduce the two models.

Computation Graphs [3]

A computation graph G is a finite directed graph consisting of: (i) nodes $n_1,n_2,\ldots,n_\ell$; (ii) edges $d_1,d_2,\ldots,d_t$, where any given edge d_p is directed from a specified node n_i to a specified node n_j; (iii) four non-negative integers A_p, U_p, W_p and T_p, where $T_p \geq W_p$, associated with each edge d_p.

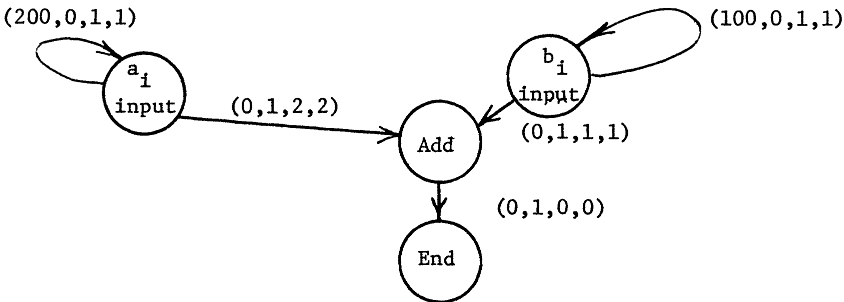
Nodes represent operations, edges represent first-in-first-out queues of data, and for an edge d_p directed from n_i to n_j the four parameters mean: A_p is the initial number of items in the queue, U_p is the number of items added to the queue each time n_i fires, W_p is the number of items removed from the queue each time n_j fires, and T_p is the number of items required as operands for n_j to fire.

The idea of computation sequences for a computation graph is that an operation can fire whenever it has a sufficient number of operands on each of its incoming edges. After firing it places results on each of its outgoing edges, and these results may be used later as operands for other operations. Some simple examples of computation graphs are shown in the following figures:



$$a_k = a_{k-1} + a_{k-2}$$

This one node, one edge example shows how a computation graph can realize the computation of successive Fibonacci numbers when the two initial values are each 1. Note here that although T = 2 here W = 1 so that two operands are needed for the add operation but only one is removed. Thus the second operand in one firing becomes the first operand in the next firing.



This example illustrates combining two lists of numbers A and B to form a list C according to the equation

$c_i = a_{2i-1} + a_{2i} + b_i$ for i = 1,2,...,100.

More complex computation graphs are shown in [3 & 5], where they are studied in more detail.

<u>Petri Nets</u> [6]

Petri nets have become a very popular means of representing parallelism in systems. Some examples of the literature are [1, 2, 6, 7]. It is a simple directed graph model in which the notion of computation can be easily explained. More formally, a <u>Petri net</u> $P = (\Pi,\Sigma,R,M_0)$ consists of a finite set of nodes Π of <u>places</u>, a finite set of nodes Σ of <u>transitions</u>, a relation $R \subseteq (\Pi\times\Sigma) \cup (\Sigma\times\Pi)$ which indicates directed edges between nodes, and a mapping M_0 from Π to the set of non-negative