COMPUTER HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS: M. Breuer, R. Hartenstein (eds)
North-Holland Publishing Company

STATISTICAL STUDIES OF HORIZONTAL MICROPROGRAMS

Peter Marwedel

Institut für Informatik und
Praktische Mathematik
der Christian-Albrechts-Universität
D-2300 Kiel 1, W-Germany

Microprograms for highly parallel machines have statistical properties different from machine-level programs. The paper discusses properties which are relevant to the design of microarchitectures. We quantify the speedup by microinstruction pipelining, the speedup by different timing mechanisms and give some guidelines for the implementation of condition logic. A software system for the design of digital processors, based on a computer hardware design language, is used to find these statistical properties.

1. Characterization of the Considered Architectures

This article discusses various statistical properties of microprograms. Although the analysing method can be used for any microprogram, we wish to emphasize those properties which can be used in the top-down design method for the class of microarchitectures we want to design [1,2].

We want to design optimized microarchitectures having the following properties:

a. Highly-parallel: Our architectures use many function boxes, memory ports and data paths in order to obtain a significant speedup over conventional architectures. These hardware units are used by powerful microinstructions which are able to control complex operations. Data may pass through several ALU's and memories before a result is stored and the next instruction is fetched. There is only one instruction stream and therefore the fetching of the next instruction is a possible synchronization point for loops, jumps and procedure calls. Therefore we have a high-speed data-flow like asynchronous operation within one instruction and, at the same time, a cheap synchronization mechanism.

b. Direct encoding [3]: The microinstruction controls the hardware units without using a lot of decoders. Only in a few cases is the microinstruction decoded. It has been shown [4] that the code size of such microprograms need not be larger than equivalent conventional machine code if the microarchitecture is carefully designed. Microarchitectures, which are designed for user microprogramming, are not harder to program than conventional machines.

c. Variable duration of instructions: We assume that not all the microinstructions are completed within the same amount of time. This is especially true for complex microinstructions. Simple architectures use a constant amount of time for all instructions, which must be the 'worst-case' time. We assume that instructions may be of different length. This length can either be computed at compile time

and stored in a field of the microinstruction or there may be tokens, which accompany the data, and specify at what time the data is valid. In the latter case, a microstep is finished if all tokens reached their destination. There are at least three reasons why architectures using token logic are faster than architectures with compiled instruction length:

α. The length of the instruction may depend upon conditions, e.g., an instruction may have a fast THEN and a slow ELSE case.
β. Parallelism for this kind of architecture needs a multi-port data memory. When large multi-port memories are realized by interleaved memory banks, the minimum execution time of an instruction depends upon the number of memory access conflicts. For a compiled instruction length we had to assume the worst-case access conflict, i.e. all operands are within the same bank. This would result in no speedup by interleaving.
γ. There may be data-dependent execution times of certain functions, like 'shift until mantissa is normalized'.

The token logic is similar to the data-flow concept but needs less overhead.

d. Complex condition hardware: In hardware, there may be two kinds of conditions:

a. a conditional write operation into a memory (i.e. either an operand is written or nothing is written),
b. the conditional selection of an operand.

Hardware for both cases is shown in Fig. 1.1. Simple additions have to be made if token logic is used. Further additions allow multiple nested conditions [5].

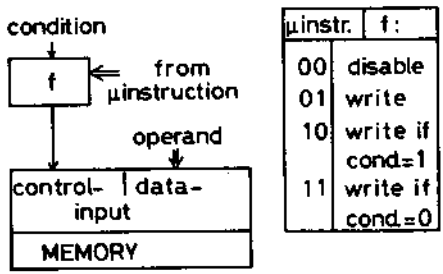

Fig. 1.1a Evoke condition

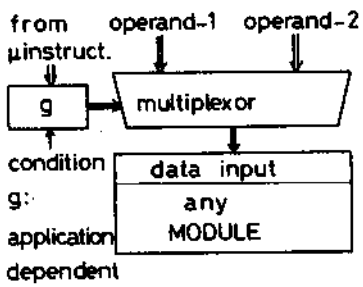

Fig. 1.1b Select condition

The microinstruction bits in Fig. 1.1.a are evoke bits, i.e. control state changes [6]. The microinstruction bits in Fig. 1.1.b are select bits, they choose a function or configure the data paths, but do not modify memory.

2. Method Used for Statistical Studies

Section 4 of this paper contains statistical figures which serve as

guidelines for the design of details of the described class of architectures. The design follows our general design methodology. It requires that all design decisions should be based upon objective criteria. At present we consider cost and performance as design criteria. Our MIMOLA-Software System (MSS) helps the designer to find cost/performance-optimized digital systems.

The MSS accepts our computer hardware design language MIMOLA [7] as its input. MIMOLA has dialects for the (procedural) description of algorithms and for the (non-procedural) description of computer hardware structures.

The MSS first converts input algorithms into data-flow trees (c.f. Fig. 2.1 and Fig. 2.2). These trees contain sources and destinations of the flow of information, arithmetic, logic and sequencing operations. The allocator part of the MSS is able to assign hardware resources to the nodes of the flow tree. The amount of available hardware resources may be limited or unlimited. The compiler part of the MSS is able to split parallel parts of the flow graph such that no resource conflicts occur. The hardware resources are declared using the hardware description dialect of MIMOLA. This declaration is known to the allocator and compiler. The MSS expects that the input algorithms are highly parallel and uses the compiler to adapt the degree of parallelism to the available hardware.

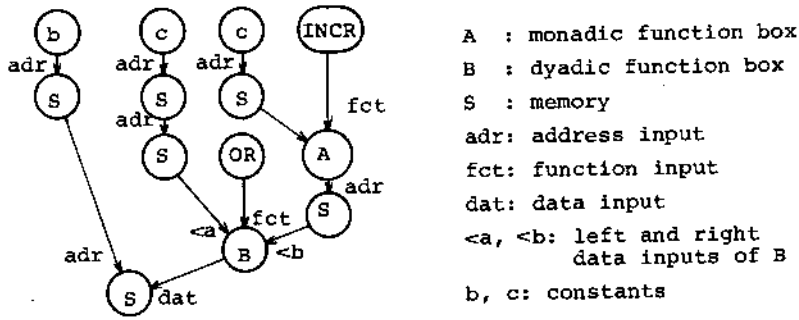

Fig. 2.1 Flow graph

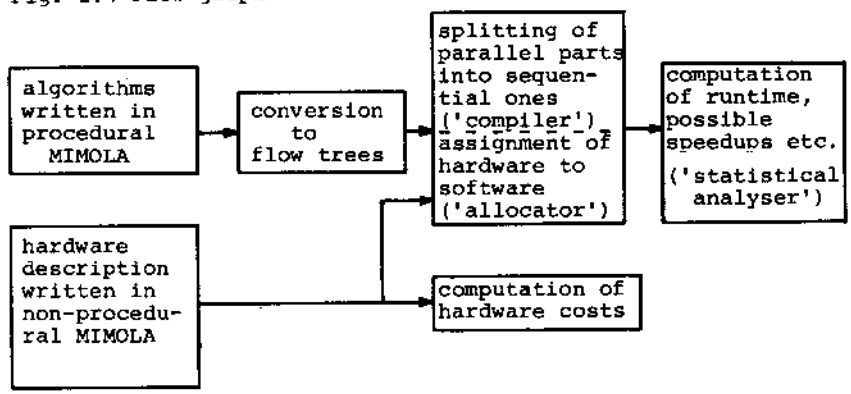

Fig. 2.2 MIMOLA Software System (simplified)

The main purpose of the MSS is to find cost/performance relations within the design space. To this end the MSS computes the cost of the used hardware and the runtime of a large set of input algorithms, the so-called test-set. Cost computation in the MSS assumes a hardware realization with discrete TTL circuits [8]. Runtime is computed by computing the minimum duration of every instruction in the test set and multiplying by the number of times each instruction is executed.

$$\text{runtime} = \sum_{\text{all instructions}} \text{duration of instruction} \times \text{iteration count}$$

Iteration counts are measured or computed outside the MSS in order to avoid long simulation runs. They are inserted into the test set and therefore are known by the MSS.

Computation of instruction durations is faciliated by the flow graph. For each of the nodes there is an associated time delay. The longest (slowest) path through the flow graph corresponds to the minimum instruction duration for architectures with compiled instruction durations. Throughout this paper we assume that time delays are equal to one time unit. This may be the time which is required to add two words. Only for nodes corresponding to the condition logic in Fig. 1.1 do we assume shorter times: Flow graphs corresponding to Figs 1.1a and 1.1b are shown in Fig. 2.3.

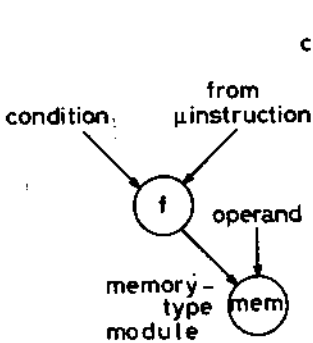

Fig. 2.3a Evoke condition

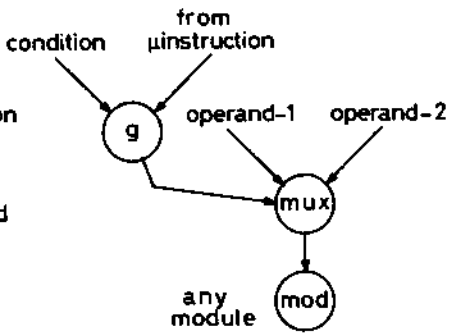

Fig. 2.3b Select condition

The graph in Fig. 2.3a must be stable after t_{ev} time units:

$$t_{ev} = \max\left(t_{op}\ ,\ (t_f + t_{cond})\right) + t_{mem}$$

where t_{op} : time when operand (and address) is valid,

t_{cond}: time when condition is valid,

t_f : delay time of network f,

t_{mem} : memory data-hold time.

The output of the module in Fig. 2.3b is valid after

$$t_{sel} = \max\left(t_{op1}\ ,\ t_{op2}\ ,\ (t_{cond} + t_g)\right) + t_{mux} + t_{mod}$$

time units, where t_{op1} : time when first operand is valid,

t_{op2} : time when second operand is valid,

t_g : delay time of network g,

t_{mux} : delay time of multiplexor,

t_{mod} : delay time of module mod.

For the token logic, the average value of t_{sel} is possibly smaller. We have

$$t_{sel,t} = \max\left(\sum_i (p_i \cdot t_{op_i}) \;,\; (t_{cond} + t_g)\right) + t_{mux} + t_{mod}$$

where p_i is the probability of using operand t_{op_i}.

A realization of f, g and mux, which is also applicable for nested IF-conditions [5], requires twice the delay time of a multiplexor for t_f, the delay time of a decoder for t_g and the delay time of a multiplexor for t_{mux}. As a first approximation, the runtime analyser sets these times by default to:

t_f = 0.5 time units,

t_g = 0.25 time units,

t_{mux} = 0.25 time units.

Besides merely computing the total runtime, the runtime analyser is able to gather other statistical data which is relevant to the design of hardware details. The analyser computes the average nesting level of IF-conditions, the minimum bandwidth of input ports, data for the design of an instruction pipeline etc. Results are discussed in section 4. Results concerning the optimum number of memory ports, function boxes etc. are contained in [8]. This paper discusses matching the structure of algorithms and some details of the structure of hardware which were left unspecified in that paper.

Structures of parallel algorithms have been studied in numerous papers by Kuck et al. For references see [9]. However, Kuck's hardware model is different from ours. It assumes a higher-level type of parallelism, not the low-level type which is provided by microinstructions.

3. Sample Programs

Five sample programs are used to gather statistical data. We also refer to these programs as the test set. The programs represent different application areas and source languages and some are written for special architectures. All programs (or 'algorithms') have been translated to a register-transfer subset of the computer-hardware design language MIMOLA. Insertion of intermediate results [9] was used in order to increase parallelism, but loops and procedure calls were not modified. Programs 1 to 3, which were not written for special architectures, will normally be compared only after a transformation to an identical hardware by the MSS compiler. These transformations generate different versions of programs 1 to 3. Only one version of each program will be used if we average over the programs. These versions will be indicated by an asterisk in Table 1. The functional capability of the test set is equivalent to about 10 k instructions of the PDP-10.

The following is a short description of the test set:

1. Part of the MSS

The MSS is written in PASCAL and makes heavy use of PASCAL's data structures. Therefore this example shows very complex addressing schemes and the degree of parallelism is very high. Program 1a is a part of the PASCAL runtime system written in MIMOLA.

2. Part of the IBM Scientific Subroutine Package (SSP)

The SSP is written in FORTRAN and uses a simple addressing scheme. Its MIMOLA equivalent does not use the maximum possible parallelism.

3. Operating System BSM [10]

The kernel of the operating system was translated to MIMOLA. The MIMOLA version contains many conditional jumps but is 'sufficiently' parallel.

4. CHILL processor [11]

The CHILL processor has a P-Code like instruction set. The microinstructions for the most common instructions have been converted to MIMOLA. The underlying hardware was not changed.

5. SIEMENS 7.760

These are the microinstructions of a redesigned SIEMENS 7.760 machine [12]. They contain a large degree of parallelism and complex condition testing.

4. Results

Table 1 contains an excerpt of the data computed by the MSS. It will be discussed in this section.

4.1 Microinstruction-Pipelining

Quite often, the next microinstruction address is known before all other expressions in a microstep are computed. At the time when the next address becomes valid, we may clock the microprogram counter and start reading the microinstruction memory. The present microinstruction is kept in a function register until all operations in the current instruction are done and the next instruction is valid. See Fig. 4.1 and 4.2 .

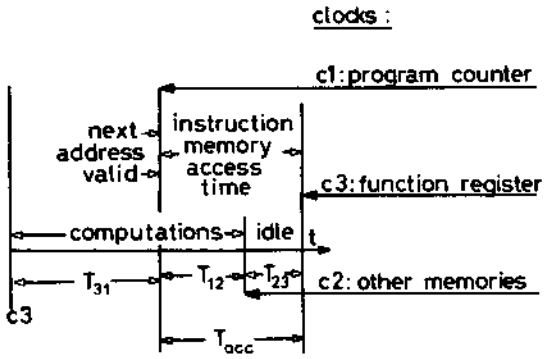

Fig. 4.1 Pipeline Timing

Algorithm	Hardware limited to	multiple use of hardware in conditions	used for averaging	$\overline{T_{32}}$ (compiled length)	speedup by compiled length	speedup by token logic	runtime (compiled length) $\overline{T_{23}} = 1$	condition nesting level N max	condition nesting level N ave	condition nesting level N var	cost $	memory input bandwidth B max	memory input bandwidth B ave	memory input bandwidth B var
1	-	yes		7.43	1.840	1.252	35217	4	0.98	0.54	9292	12	1.03	1.27
1	-	no		7.35	1.378	1.252	34892	4	0.98	0.54	9009	12	1.01	1.25
1	DCL.BSM	yes	*	4.23	1.816	1.066	95588	3	0.22	0.46	4740	2	0.70	0.64
1	DCL.BSM	no		4.22	1.822	1.067	96025	3	0.22	0.46	4483	2	0.70	0.63
1a		yes	*	5.42	1.401	1.041	117	7	1.00	2.04		4	1.22	1.02
2	-	yes		4.56	1.797	1.069	17619	1	0.70	0.46		5	0.81	0.58
2	DCL.BSM	yes	*	3.78	1.675	1.033	30366	1	0.37	0.48		2	0.72	0.43
2 a	SPDM	yes	*	4.04	1.487	1.003	90998	1	0.29	0.45		3	0.93	0.48
(different subset of SSP)														
3	-	yes		3.37	2.345	1.087	64792	2	0.59	0.54	8757	13	1.28	1.34
3	-	no		3.31	1.973	1.080	63859	2	0.59	0.54	7343	13	1.22	1.18
3	DCL.BSM	yes	*	3.19	2.269	1.040	94839	2	0.39	0.51	3346	3	0.95	0.60
3	DCL.BSM	no		3.19	1.790	1.024	97107	2	0.37	0.49	3254	3	0.90	0.59
4	CHILL-Proc.	no	*	3.13	1.453	1.000	17766	1	0.50	0.50		1	0.56	0.50
5	7.760	yes	*	3.32	1.738	1.016	413998	2	1.26	0.68		1.6	0.52	0.34

Table 1

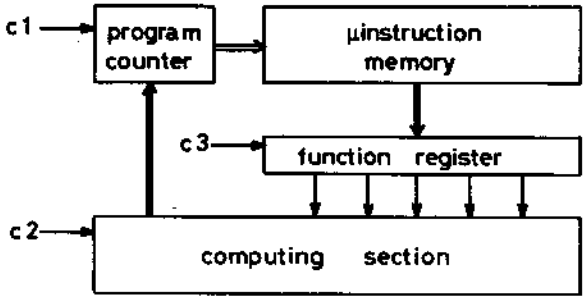

Fig. 4.2 Pipeline hardware

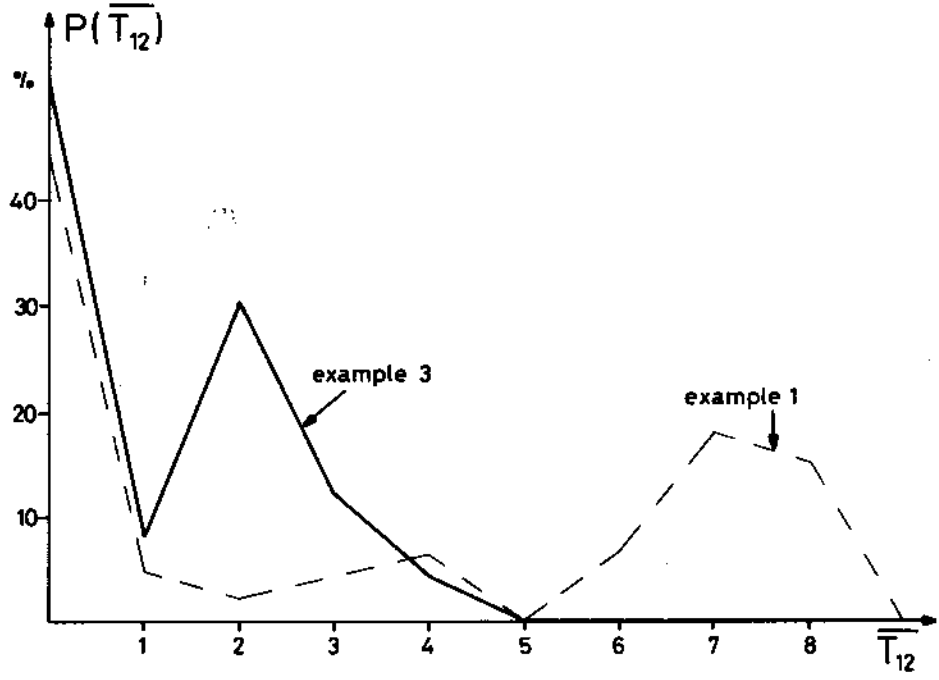

Fig. 4.3 Probabilities $P(\overline{T_{12}})$

The clock C1 may be derived by valid-tokens accompanying the data. The MSS analyser allows us to study the frequency distributions of the time-intervals T_{12} between clock pulses C1 and C2 for our sample programs. These distributions differ from program to program. Fig. 4.3 shows the most extreme distributions of our test set.

Example 1 allows a good pipelining. This is due to the complicated expressions in this example combined with the PASCAL addressing scheme of variables. Only few jumps depend on multiple nested conditions. On the other hand the operating system uses simple expressions, a simple addressing scheme and many conditional jumps.

The distributions of the intervals T_{12} can be used to find the relation between the average time $\overline{T_{23}}$ between clocks C2 and C3 and the instruction memory access time T_{acc}:

$$\overline{T_{23}}\ (T_{acc}) = \sum_{T_{12}=0}^{T_{12}=T_{acc}} P(T_{12}) \cdot (T_{acc} - T_{12})$$

where $P(T_{12})$ is the probability of a certain time interval between clocks 1 and 2.

Fig. 4.4 shows $\overline{T_{23}}$ for some program versions of our test set.

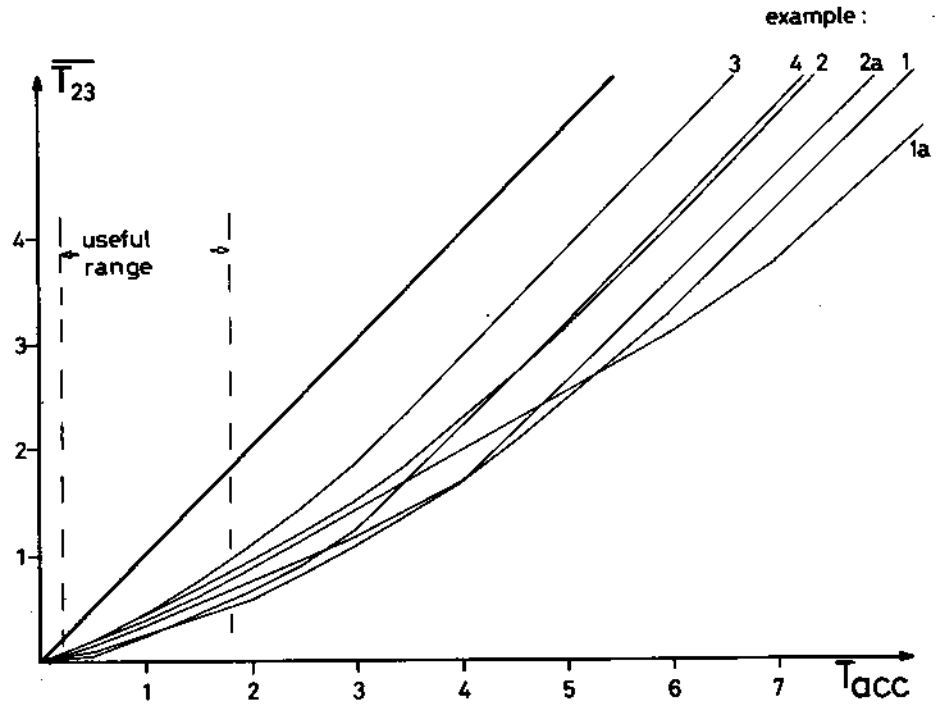

Fig. 4.4 Relations between memory access and idle time

The straight line corresponds to the no-pipelining case, i.e. $\overline{T23} = T_{acc}$

The average speedup by pipelining for our algorithms is

$$S_p = \frac{1}{n} \sum_{i=1}^{n} \frac{\overline{T_{32_i}} + T_{acc}}{\overline{T_{32_i}} + \overline{T_{23_i}}}$$

where $\overline{T_{32_i}}$ is the average duration of an instruction without instruction fetch for algorithm i and $\overline{T_{23_i}}$ is the average 'idle' time.

Average instruction durations ($\overline{T_{32}}$) are listed in Table 1. Values range from 3.13 to 7.43 . A value of 3 corresponds to reading a scratch-pad memory (1 time unit), arithmetic or logic manipulation (1 time unit) and storing the result (1 time unit). Therefore, a value of 7.43 indicates very high expression trees. Expression tree height reduces considerably if the number of hardware resources is limited.

Using $\overline{T_{32}}$ of the lines in Table 1 which are marked by an asterisk and $\overline{T_{23}}$ of Fig. 4.4 (setting $T_{acc} = 1$), we obtain an average speedup of

$$S_p = 1.125$$

The average access time for the microinstruction memory may also be reduced if, at the start of each instruction, a fetch of the instruction with the next higher address is started ('prefetch'). This prefetch will be valid if no jump occurs. The MSS computes jump probabilities for the test set ranging from 25 % to 50 %, resulting in a prefetch speedup of about 10 %. This speedup is a guideline for the designer who wants to know if he should implement the prefetch logic.

4.2 Speedup by Variable (compiled) Duration of Microsteps

If we compare architectures with fixed and with variable (compiled) duration of microsteps, the maximum speedup is (i ranging over all instructions):

$$S_v = \frac{\max_i (\text{duration of instruction } i) * \sum_i \text{number of times } i \text{ is executed}}{\sum_i (\text{duration of instruction } i * \text{number of times } i \text{ is executed})}$$

Table 1 shows the speedup for various forms of the algorithms. The speedup is high if there is a small number of slow microinstructions and a large number of fast microinstructions. The speedup may be as large as 2.345 . Using only the forms indicated by an asterisk, we have an average speedup of

$$S_v = 1.69$$

4.3 Speedup of Conditional Statements by Tokens

In section 1 we mentioned the speedup of conditional statements by tokens. If we use formula (1) for the computation of runtimes, we obtain an average speedup of

$$S_c = 1.028$$

using again the versions indicated by an asterisk. This small quantity is not sufficient by itself to justify the additional hardware.

But the main reason for a token logic is the interleaved memory and the speedup of 1.028 for conditional statements is only a byproduct.

4.4 Average Number of Nested Conditions

This number is defined as (i ranging over all instructions)

$$N = \frac{\sum_i (\text{maximum condition nesting in } i) * (\text{number of times } i \text{ is executed})}{\sum_i (\text{number of times } i \text{ is executed})}$$

N is about 0.8 for the most parallel programs in the test set and about 0.3 for examples with a limited number of memory ports and operators. This indicates that the implementation of one condition level is sufficient. Only program 5 needs 1.26 condition levels on the average, this is due to testing for valid data in the cache, in the address translation memory, in the main memory and many error conditions.

4.5 Multiple Use of Hardware in Conditions

Within one instruction, the same hardware unit may be used for different purposes under mutually exclusive conditions. E.g. a memory port may read cell 1 in the THEN-part and cell 2 in the ELSE-part. An option of the MSS-allocator is available which disables such a multiple use of hardware. We studied the effects of this option for programs 1 and 3, see Table 2. All values are normalized to one if the multiple use is not allowed.

Example No.	Number of ports	relative runtime	relative cost	runtime times cost
1	unlimited	1.009	1.031	1.041
1	8	0.995	1.057	1.053
3	unlimited	1.014	1.193	1.210
3	8	0.977	1.026	1.002

Table 2 Effect of allocator option

Multiple use of hardware increases speed, if a fixed number of ports is given, but decreases speed, if the number of ports is unlimited. The latter is due to an reduced overlapping in the computation of the condition, the THEN-part and the ELSE-part. Multiple use of hardware increases cost due to more connections and microinstruction fields. But the effect on the cost-runtime product is so small that only a more detailed analysis is able to find out the best solution.

Multiple use of hardware leads to supplying memory ports with more than one microinstruction address field. What we want to point out here is that this can be cost-effective.

4.6 Bandwidth of Memory Inputs

If at the end of the microsteps the results are stored, some of these possibly have to be stored in the same memory bank. This conflict causes a delay. It can be minimized if all results are stored as early as they are known. Based upon the knowledge of the earliest possible arrival times of the results, the runtime analyser computes the minimum number of results the memory must be able to store within a time unit if no conflicts shall occur. We call this the minimum

bandwidth B. This bandwidth was averaged over our sample programs and is shown in Table 1.

4.7 Critical Path Analysis

Slow modules, which are bottle-necks of the whole hardware system, may be found by a critical path analysis. To this end, the runtime analyser computes the frequency of finding a module in the critical (i.e. longest) paths of the flow trees. The designer may replace the modules, which are found frequently on the critical paths, by faster modules.

Conclusion

The paper presents a method which may be used in order to gain more insight into the time behaviour of microprogrammed, low-level parallel algorithms. It shows that register-transfer and computer-hardware description languages may be successfully applied in order to compute figures upon which hardware design decisions may be based. The final design slightly depends upon the main application area of the processor. However, the implementation of a variable instruction duration seems to be valuable in most cases and that the implementation of nested conditions in hardware is normally not necessary.

References

[1] Zimmermann, G., Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA, Informatik Fachberichte 5 (1976) 465-478

[2] Zimmermann, G., The MIMOLA Design System: A Computer Aided Digital Processor Design Method, 16th Design Automation Conf. Proc. (1979) 53-58

[3] Agrawala, A.K., Foundations of Microprogramming (Academic Press, New York, 1976), p. 62

[4] Marwedel, P., The Design of a Subprocessor with Dynamic Microprogramming with MIMOLA, in: Zimmermann, G. (ed.), Informatik Fachberichte Bd. 27 (Springer, Berlin, 1980)

[5] Zimmermann, G., Private communication, 1980

[6] Nagle, A.W., Automatic Synthesis of Microcontrollers, 15th Design Automation Conf. Proc. (1978) 112-117

[7] Marwedel, P. and Zimmermann, G., MIMOLA Report Revision 1 and MIMOLA Software System User Manual, Techn. Rep. 2/79, Institut für Informatik und Prakt. Mathem., Kiel Univ. (May 1979)

[8] Zimmermann, G., Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down Design Method, Proc. 4th Int. Symp. on Computer Hardware Description Languages (1979) 33-39

[9] Kuck, D.J., The Structure of Computers and Computations, Vol. 1 (Wiley, New York, 1978), p. 111

[10] Entwicklungsbeschreibung BSM, Rechenzentrum der TU München (1972)

[11] Leisch, F., The Impact of High-Level Languages on the Architecture of a microprogrammed HLL Microcomputer, IMM/Datacomm '80 (Genf, 1980)

[12] Krüger, G., Entwurf einer Rechnerzentraleinheit für den Maschinenbefehlssatz des Siemens Systems 7.000 mit dem MIMOLA-Rechnerentwurfssystem, Diploma Thesis, Institut für Informatik und Prakt. Mathem., Kiel Univ. (June, 1980)