The Design of a Subprocessor with Dynamic Microprogramming with MIMOLA

Peter Marwedel
Institut fur Informatik and
Praktische Mathematik
Olshausenstr. 40-60 2300 Kiel
1   W.-Germany

Abstract

MIMOLA is a language for the optimized design of digital processors, based upon computing resource utilizations for typical programs. It has been used for the design of a well-structured, fast, parallel and microprogrammable processor. Although not larger than a conventional minicomputer, it is about 26 times faster. It proves, that microcode need not be larger than equivalent machinecode. This paper also discusses possible architecture alternatives with low cost/performance ratios.

O. Introduction

MIMOLA is a language for the top-down design and description of computer hardware.

The conventional computer aided design starts with a description of a computer structure /Bre/. The design system then faciliates the step-wise refinement of the design but it does not explore the cost/ performance tradeoffs in the design space. In conventional systems, simulations measure performance only roughly. Simulation times are rather high (for that reason, CASSANDRE was extended by LASCAR /Bor/). Therefore only few sets of input parameters can be tested. Furthermore, many systems use non-procedural languages (i.e. languages without implicit control flow like e.g. CDL /Chu/). It is hard to write a large number of programs without implicit control flow. Therefore only a small number of programs is simulated and the result is an unreliable performance estimate. Therefore the new language MIMOLA was defined /Zim 77/.

1. Designing with MIMOLA

MIMOLA allows us to estimate performance and cost of digital structures. It is a procedural language (i.e. the flow of control is simi lar to conventional programming languages. ike PASCAL). This makes it possible to write such a large number of algorithms in MIMOLA that reliable performance estimates can be derived-For this purpose MIMOLA is problem-oriented. But at the same time it is easy to give software statements a hardware meaning. One of the tools for this is a macro processor /Hol/. Any sequence of MIMOLA symbols, that can be reduced to a single symbol, can be replaced by any other sequence of symbols. High-level language (HLL) elements like e.g. FOR .. TO, WHILE .. are special cases of macros. The processor allows parameters and considers the types of parameters. This flexible macro mechanism is missing in HDL /Hof/.

After the HLL elements have been replaced, part B of our MIMOLA Software System (MSS)/MaZ/ is able to map all language expressions into hardware /Mar/. Cost computation is based on the fact that a structured hardware description (consisting e.g. of modules and ports) is generated. Hardware may but need not be predeclared. The declaration allows to put a limit on the resources that can be used by a program. Intermediate steps are automatically inserted if this is a way to get round hardware limitations /UZi/. By insertion of intermediate buffering and new jumps, original semantics remain unchanged.

After the software/hardware mapping the MSS computes cost/performance characteristics and utilization statistics. This includes for example the first and second order utilization frequencies of modules, ports and the fields of the microinstruction. Instead of computing repeat counts for microsteps by time-consuming simulations,the MSS uses a different and new approach: the user may manually set weighting factors for the microsteps. In a number of cases it is easy to determine these factors: In matrix computation they are normally equal to a power of the matrix dimension, for sorting and searching statistical methods may be used /Knu/ and for operating systems hard- or software monitors measure statistics more realistic than simulators.

A MIMOLA design starts with a set of typical application programs for the target processor /Zim 76, Zim 79a/. The MSS maps these-programs into hardware and computes utilization statistics. A large amount of hardware is usually required due to some highly parallel parts of the input program. The resources will be used inefficiently and the designe may limit the resources to those declared in a hardware declaration. The MSS will do those transformations on the input program that are necessary in order to make the programs executable on a limited hardware. Statistical analysis of the transformed programs allows the repetition of the process until a good cost/performance relation is obtained.

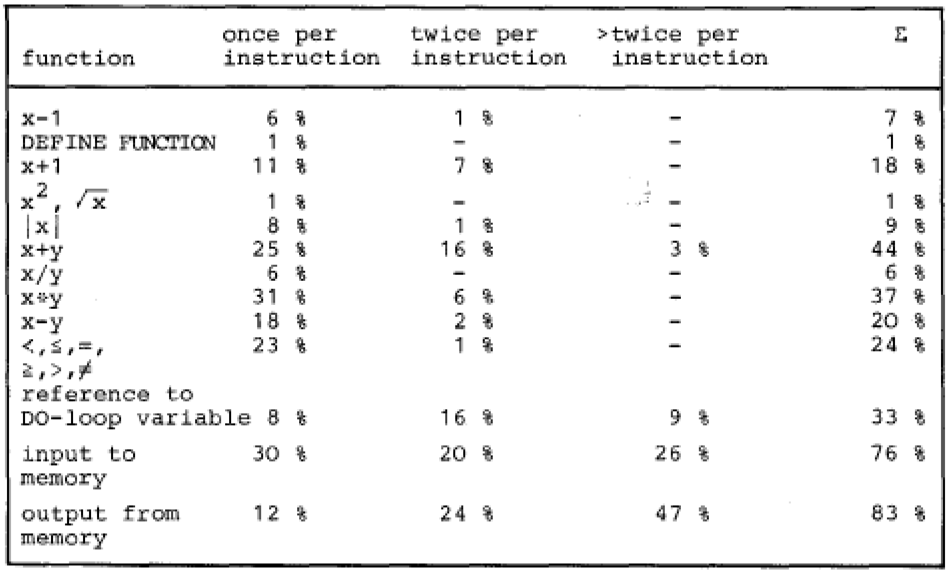
2. Design and Implementation of the SPDM /KSc,WSc/

After the definition of the design method it had to be applied to a real problem. The large speed of highly parallel, microprogrammable computers had already been proven by an older design /Zim 75/. This was a special purpose processor that was coupled through a fast DMA-channel to a general purpose minicomputer and therefore the processor was called a subprocessor with dynamic microprogramming (SPDM). This asymmetric double-processor structure speeds up lengthy computations by a significant factor while there is no need to design a new operating system with assembler, compilers and editor.

It was decided to apply the MIMOLA design method to systematically design a SPDM, using the IBM Scientific Subroutine Package (SSP) /MOD/ as a representative for scientific calculations. Fortunately the SSP does not contain I/O-statements and only a low number of called routines. Therefore we can load all called subroutines into the SPDM and need not interrupt the execution process until a complete SSP subroutine has been executed.

18 subroutines of different mathematical areas of the SSP were selected because we assumed that 18 routines 'provide a sufficient statistical basis. The SSP is written in FORTRAN, this allows an easy translation of the SSP to MIMOLA. Sequential FORTRAN flow was translated into parallel MIMOLA microsteps. At an average, between two and three FORTRAN statements were put into a single microinstruction step (c.f. /Kuc/). Conditional FORTRAN statements were translated to IF .. THEN .. ELSE .. FI . Nested conditions within one microstep were not used.

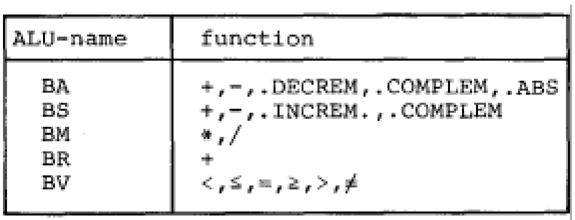
These 18 routines were then analysed by a preliminary version of the MSS (allowing only statistical analysis). Table 1 shows how often certain function or storage modules are required for the initial MIMOLA version.



function	once per instruction	twice per instruction	>twice per instruction	Σ
x-1	6 %	1 %	-	7 %
DEFINE FUNCTION	1 %	-	-	1 %
x+1	11 %	7 %	-	18 %
x^2, $\sqrt{x}$	1 %	-	-	1 %
$\lvert x \rvert$	8 %	1 %	-	9 %
x+y	25 %	16 %	3 %	44 %
x/y	6 %	-	-	6 %
x*y	31 %	6 %	-	37 %
x-y	18 %	2 %	-	20 %
<,≤,=, ≥,>,≠	23 %	1 %	-	24 %
reference to DO-loop variable	8 %	16 %	9 %	33 %
input to memory	30 %	20 %	26 %	76 %
output from memory	12 %	24 %	47 %	83 %

Table 1. Function and memory statistics for initial program version

This table shows that there should be at least 3 memory ports in each direction in addition to the DO-loop index storage. Large memory circuits with more than an input and an output port are not available. However, a lot of memory references is to a low number of local scalar variables. Therefore, a separate small multi-port memory can reduce the load on the main memory. With the integrated circuits SN 74172 two input- and two output-ports (with one common address) can be easily implemented.



ALU-name	function
BA	+,-,.DECREM,.COMPLEM,.ABS
BS	+,-,.INCREM.,.COMPLEM
BM	*,/
BR	+
BV	<,≤,=,≥,>,≠

Table 2. Function boxes of SPDM

The mapping of functions to function boxes was based upon the second order frequency distribution of used functions. This led to five function **boxes** (c.f. Table 2). Other functions are implemented as subroutines.

The 18 input routines were then transformed by hand such that they fitted into this limited hardware and the MSS statistical analyser computed utilization statistics for modules and connections. Modules were now used more efficiently. However, there was a large number of infre quently used connections. The number of connections was reduced by: a) replacing shift operations by multiplications and divisions b) replacing a ROM used for constants by a reference to a preloaded RAM c) replacing DO-loop counters by a second multi-port RAM d) replacing memory to memory transfers by an addition of zero.

Table 3 is a comparison of the hardware requirements and the estimated runtime for the three mayor design steps.

	initial programs	reduction of modules and ports	final design
# of microinstructions	480	728	1015
µ instruction word and length	>220	150	112
# of connections (16 bit each)	394	140	100
estim. run-time for computation of eigenvalues matrix dimension = 32×32 [msec]	18.6	19.4	31

Table 3. Characteristics of the three MIMOLA program versions

Fig. 1 shows the block diagram of the final design.

The hardware was implemented by an experimental module system, containing for example 16 bit ALU's with look-ahead, multiplexes-units and different storage modules. Flat cables are used for 16 bit data paths. This system eases testing. The total system includes about 670 integrated circuits, most of them MSI except for the memory chips.

The SPDM (slave) is coupled to the MODCOMP II (master) by a customdesigned channel /Ber/ that has complete control over the MODCOMP's internal buses and uses the full memory speed (0.8 usec/16 bit) of the MODCOMP II. While the SPDM is busy, the channel may disable operations in the MODCOMP (except DMA-Transfers) or return the control to the master. If the control is returned to the master, the SPDM may interrupt it upon completion. Hence, parallel operation in the MODCOMP is possible.

3. Results

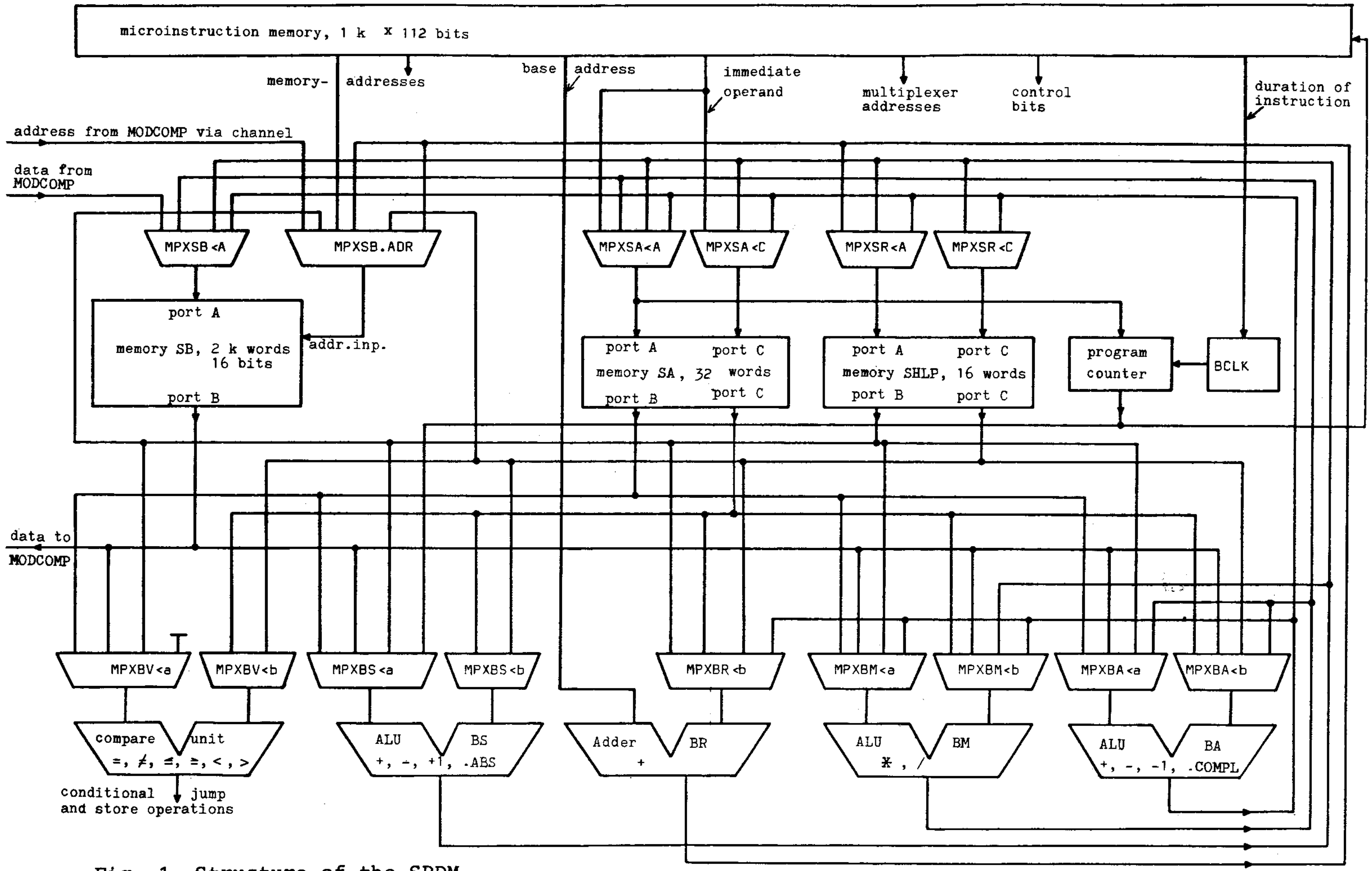


Fig. 1 Structure of the SPDM

sub-routine name	dimen-sion	load time [µsec]		execution time [µsec]		speed increase		
		pro-gram	data	SPDM	MODCOMP	exec only	with data load	with data & program load
GAMMA	-	263	2.5	13.3	675	51	43	2.4
LEP	11	69	10.4	33	825	25	19	7.5
MINV	15x15	465	362	19200	871000	45	44.5	43.5
SIMQ	15x15	320	206	8600	156000	18.1	17.7	17.1
MATA	5x 5	220	42.4	488	7100	14.5	13.4	9.4
	15x15	220	362	8388	150400	18	17.2	17.1
RANK	10	130	17.6	232	3900	16.8	15.6	10.2
	100	130	161	2949	70600	23.9	22.7	21.8
Average						26.5	24.1	16.1

Table 4. Speed of SPDM

This table shows that the SPDM increases the speed by a factor of about 29 if the program is preloaded in the SPDM and by about 16 if it is not. This is about the factor the floating point unit speeds up floating point operations. The SPDM is far more flexible than a floating point unit and costs are not very different if memory prices keep going down.

SPDM-execution times are calculated for GAMMA, LEP, MINV, SIMQ and measured for MATA and RANK.

Some people believe that the codesize of microprograms is larger than the codesize of equivalent machine language programs. As can be seen from table 5 this is not true for the SPDM.

Subroutine name	Code SPDM hand-translated (= 100 %)	Code MODCOMP II Compiler FRX Rev M.00 abs.	rel.	PDP-10/KI Compiler FORTRA V5 abs.	rel.
CANOR	12320	14128	115 %		
MATA	2576	2672	104 %	2952	115 %
MINV	9296	10992	118 %		
MULTR	6048	6576	109 %		
SIMQ	6384	6656	104 %		
Σ	36624	41024	112 %		

Table 5. Comparison of Codelength [bits]

First experiments with automatic code-generation with the aid of the MSS resulted in about 10 % more code compared to hand-translation. Therefore the SPDM needs about as much code as the MODCOMP II, even if suboptimal code-generation is used.

For further results we use the MSS instead of the real SPDM. This has the following advantages:

- Statistical results can be obtained with the statistical analyser of the MSS more easily than with a logic state analyser
- The time characteristics of the hardware can be changed by declaration. This allows us to study the effects of memory access times, multiplication times etc.
- The structure of the hardware can be changed.

However, we have to guarantee that the repeat counts for the statements and the time that is required to execute a statement are computed correctly. Therefore (and in order to speed up MSS analysis) we choose a subset of five matrix routines of the SSP with repeat counts equal to a power of the matrix dimension. The MSS computes a total execution time for subroutine MATA that is 20 % larger than the value measured by Schulz and a total execution time for MINV that is 18 % shorter than the time computed by Schultze. Differences are due to incomplete time specifications for the SPDM and therefore we do not attempt to reduce them now.

As a first application of the MSS we determine how much the fast modules contribute to the speed of the SPDM. To that end we assume that we had to build the SPDM with modules that have the same speed as those in the MODCOMP. For example, multiplications need 6.42 µsec (instead of 0.3 µsec) and divisions need 10.74 µsec (instead of 0.6 µsec). Reading and writing from or to the main (core-) memory requires between 0.3 µsec (access-time) and 0.8 µsec (cycle-time) in the MODCOMP. Using 0.3 µsec for the read access-time and for write data hold-time the SPDM would be 5.37 times slower than the actual SPDM, using 0.8 µsec it would be 7.31 times slower. This means that the SPDM would still be 26.5/7.31= 3.62 ties faster than the MODCOMP, even if we had to wait a full memory cycle for every read and write operation.

Obviously the precise result depends on the amount of multiplications and divisions in the test program. Furthermore we assumed that a full microinstruction is read from main memory in one memory cycle. One cannot completely seperate the influence of the architecture and the module speed because the MODCOMP has seperate memories faith different access times for two kinds of instructions while the SPDM has only .one instruction memory.

We conclude that the speed increase caused by the architecture is about the same order of magnitude as the speed increase caused by the fast modules.

4. Design Alternatives

As a second application of the MSS we examine how effective the microinstruction of the SPDM is used (Fig. 2)

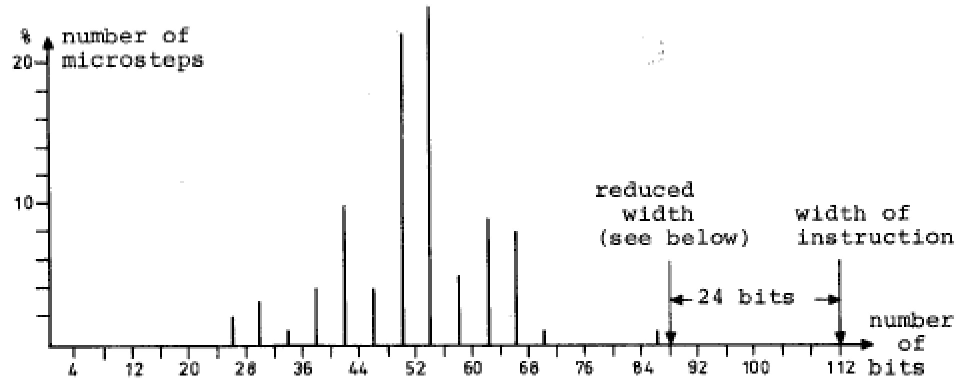


Fig.2 Distribution of the number of used bits

Although the instruction length is not used very inefficiently, a reduction of the length by about 20 bits seems to be possible. For this reduction we consider the use of one instruction field for two controlled destinations. This is possible if the original field only selects a function, a memory or a multiplexer-address ('select' fields /Nag/) and does not require additional multiplexing bits. The second order frequency distribution of used instruction fields determines fields that can be put together. The fields need not be used mutually exclusive. Microsteps that need fields for both destinations are automatically split into two instructions by the MSS.

Two possible reductions have been tested: 1. Using the jump address field also for the address of memory SA port A, thereby saving five bits. 2. Using the jump address field for the address of port C of memory SA, the address of port A of memory SR for port A of memory SA and using only one direct address for the two ports of memory SB. This saves 20 bits. Table 6 shows the resultant changes in runtime, instruction width and cost (unchanged SPDM = 100 $):

	instruction width	number of instructions	run-time	total cost	cost × runtime
change 1	95.5 %	100.9 %	100.9 %	98.6 %	99.5 %
change 2	82 %	108.6 %	104.6 %	94.3 %	98.6 %

Table 6. Influence of the reduction of the instruction width

The relative reduction of the total cost is not very large because of the cost of the other modules. - Another four bits can be saved if instruction fields for multiplexers are included in the code generation. However, the current MSS cannot yet split microstatements with resource conflicts at multiplexers. But because there are only few resource conflicts for these 4 bits we may assume we can save 24 bits of 112 (21 $). Further reduction will significantly increase runtime (as can be expected from Fig. 2).

The present MSS allows us to explore cost/performance relations in the design space to a larger extend than the designer of the SPDM could. We now may compare the SPDM to other architectures, including different memories and operators and a more detailed speed analysis.

At first we analyse the requirements of the mentioned five matrix routines. After an initial pass through the MSS we limit the hardware resources mainly following the method that is outlined in /Zim 79b/ we delete all modules and ports whose ration utilization (in $) divided by cost is below 1. Instead of the absolute utilization we use the joint utilization of the resource that is to be deleted and another resource that can serve as a substitute.

For the purpose of this paper we minimize only modules and ports and do not attempt to minimize the number of connections and the width of the instruction.

We compare three design styles: 1. a processor with a small memory SHLP for intermediate buffering and a large memory SB. We start with different modules for different functions. Function boxes with more than one function **are introduced** during the deletion process if one module has to take over the job for one that is deleted. 2. same as 1. except that all local scalar variables are kept in a small memory SA. 3. same as 2. except that we start with only one type of function box that can execute all required functions.

Fig. 3 shows the resultant cost/performance relations for three combinations of the sizes of data memory SB and instruction memory SI:

a. SB : 2 K × 16 bit, SI: size of tested routines (~ 5 K bytes)
b. SB : 20 K × 16 bit, SI : 10 × size of tested routines
c. SB : 200 K × 16 bit, SI : 100 × size of tested routines

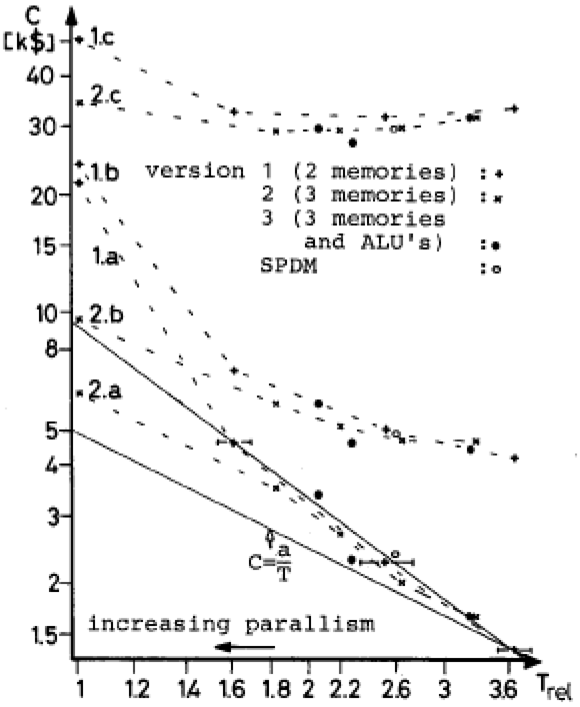


Fig.3 Cost/performance relations

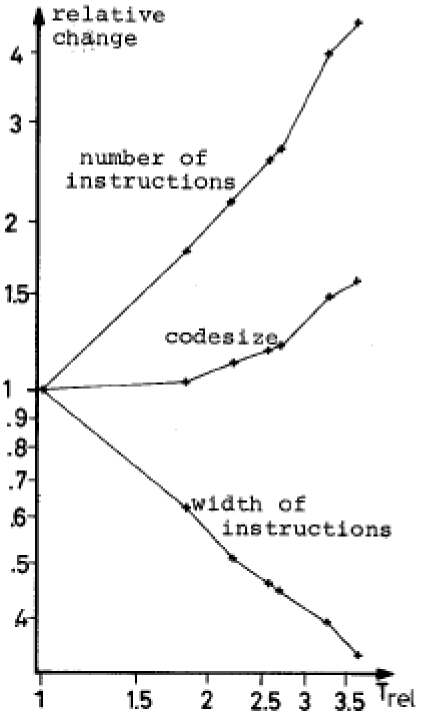


Fig.4 Change of codesize

For small memories SB and SI (.a) the right most point corresponds to the lowest speed-runtime product. For large memories (.c) the cost of SI dominates and because it decreases much slower than the runtime increases, the lowest codesize (Fig. 4) and the lowest speed-runtime product is obtained on the left side of the drawing (this means that parallel computers are best if a large number of instructions is stored). For the matrix routines the codesize (= number of instructions times instruction width) of parallel computers is even smaller than for more sequential computers (Fig. 4).

The use of a local memory reduces costs mainly for highly parallel architectures (left most points in Fig. 3) and for a large number of stored instructions (.c). In these cases the large number of addressing bits for SB increases costs siginificantly.

Multiple-function boxes (ALU's) instead of single-function modules seem to be useful in the medium range of T. For large values of T there are two multiple-function boxes for all alternatives and thus there is no difference between alternative 2. and 3. . For small values of T the ALU's lead to more connections (this partly may be caused by our present software/hardware mapping and needs further investigation).

For a. all points can be approximated by a straight line in the logarithmic diagram:

$$C = 9.2 \cdot T^{-1.49}$$

Fig. 3 also contains the actual SPDM, except that memory sizes have been computed like those for the other points of the diagram. It turns out that the SPDM is between 11 and 1 % slower than the MSS-optimized structures with equal cost. The difference would be larger if we had used instruction width and connections optimization. The SPDM corresponds to architectures with a lower number of ports than it has. This shows that the 'effective' number of ports in the SPDM has been decreased by a rigorous limitation to the number of connections.

Further studies showed that runtime decreases by about 10 % if the controlling operand of IF-clauses and the THEN-part are evaluated simultaneously. It increases by the same amount if the access-time of memory SB is doubled.

We may want to estimate how much the inclusion of additional matrix routines will affect our cost/performance relations. The cost computation is not influenced by statistical errors. In order to estimate the error of the runtime computation we consider the runtime increments of each of our five routines to be a measurement with value x_i $(1 \leq i \leq 5)$. Three error bounds, based upon the standard deviation of the x_i have been drawn in Fig. 3.

For comparison we analyse a set of eight mathematical functions of the SSP (Fig. 5).

Exept for the rightmost points, we did not combine the multiplication unit with the other function boxes although we had to do this if we had strictly used the deletion criterion 'utilization/cost < 1'. Therefore the cost-runtime product of the rightmost points, corresponding to only one ALU, is comparatively low.

Points marked by 'o' are obtained by executing the mathematical functions on the hardware that is optimized for the five matrix routines. The functions execute about 9% slower on the matrix hardware, compared with the optimized mathematical function hardware.

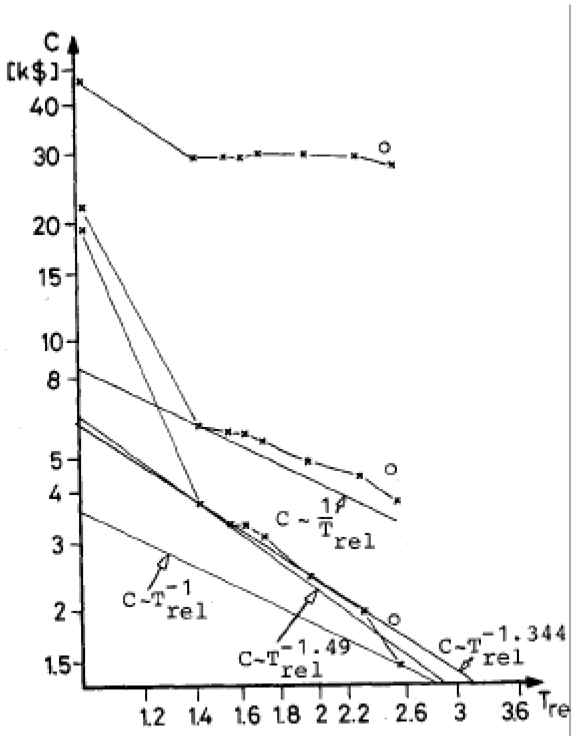


Fig. 5 Cost/performance relations for mathematical functions

Acknowledgement

I want to express my thanks to those whose work contributed to this paper: R. Berlinski (fast channel), R. Hollenbach and U. Zimmermann (MSS), K. Schultze and W. Schulz (SPDM), and G. Zimmermann (fundamental ideas).

References

/Ber/ R. Berlin ski: "Ein mikroprogrammierbarer DMA-Kanal zur Rechnerkopplung", Diploma Thesis, Kiel, 1978.

/Bor/ D. Borrione: "Description et Simulation dune architecture m8lti-Processeur a 1'aide du language LASCAR", Rapport de Recherche-n 87 ENSIMAG, Grenoble,1977.

/Bre/ Y. Bressy, B. David, Y. Fantino, J. Mermet: "A Hardware Compiler for interactive Realisation of logical Systems Described in CASSANDRE". Proc.Int.Symp. on Computer Hardware Description Languages, New York, 1975.

/Hof/ R. Hoffmann: "Rechenwerke and Mikroprogrammierung", Munchen 1977.

/Hol/ R. Hollenbach: " Ein flexibler Makroprozessor fur die Sprache MIMOLA", Diploma Thesis, under preparation.

/Knu/ D.F. Knuth: "The art of computer programming", Vol. 1, Reading, 1973.

/KSc/ K. Schultze: "Systematischer Entwurf eines Prozessors mit der Sprache MIMOLA", Diploma Thesis, Kiel, 1978.

/Kuc/ D. J. Kuck, Y.Muraoka, S.-C. Chen: "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup, IEEE _C-21 (1972), 1293-1310.

/Mar/ P. Marwedel: "The MIMOLA DESIGN SYSTEM: Detailed Description of the Software System", Proc. 16th Design Automation Conference, San Diego, 1979.

/MaZ/ P. Marwedel, G. Zimmermann: "MIMOLA Report Revision 1 and MIMOLA SOFTWARE SYSTEM User Manual", Report 2/79 of the Institut fur Informatik and Praktische Mathematik, Kiel, 1979.
/MOD/ MODCOMP Inc.: "MAX II/III Library: Scientific Subroutine Library", Ft. Lauderdale. 1976.
/Nag/ A. W. Nagle: "Automatic Synthesis of Microcontrollers", Proc. 15th Design Automation Conference, 1978.
/UZi/ U. Zimmermann: "Ein Compiler zur Sequentialisierung von MIMOLAProgrammen", Diploma Thesis, 1979.
/WSc/_ W. Schulz: "Realisierung eines Rechnerverbundes aus einem Universalrechner and einem mit einer Entwurfssprache konstruierten Prozessor", Diploma Thesis, Kiel, 1979.
/Zim 75/ G. Zimmermann: "SPDM - A Subprocessor with Dynamic Microprogramming", EUROMICRO, Nice, 1975.
/Zim 76/ G. Zimmermann: "Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA", Informatik-Fachberichte, Stuttgart, 1976.
/Zim 77/ G. Zimmermann: "Report on the Computer Architecture Design Language MIMOLA", Report 4/77 of the Institut fur Informatik and Praktische Mathematik, Kiel, 1977.
/Zim 79a/ G. Zimmermann: "The MIMOLA DESIGN SYSTEM: A Computer Aided Digital Processor Design Method", Proc. 16 Design Automation Conference, San Diego, 1979.
/Zim 79b/ G. Zimmermann: "Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down-Design Method", Proc. Int. Symp.. on Computer Hardware Description Languages and Their Applications, Palo Alto, 1979.