UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

MTAC - A Multithreaded VLIW Architecture for PRAM Simulation

Martti Forsell

Report A-1996-7

ACM C.1.2
UDK 681.3.02
ISSN 0789-7316
ISBN 951-708-476-5

MTAC - A Multithreaded VLIW Architecture for PRAM Simulation

Martti J. Forsell

Department of Computer Science, University of Joensuu, PB 111, SF-80101 Joensuu, Finland

Phone + 358 -13 - 251111
Fax + 358 - 13 - 2513290
email mforsell@cs.joensuu.fi

November 20, 1996

Abstract

Multithreaded processors are the key components for hiding the latency of long operations slowing down the execution of parallel programs, because they can execute other processes while a process is waiting for a long operation to complete.

In this paper we outline and evaluate a MultiThreaded VLIW processor Architecture with functional unit Chaining (MTAC), which is specially designed for efficient PRAM simulation. According to experiments MTAC offers remarkably better performance than a basic pipelined RISC architecture and chaining improves the exploitation of instruction level parallelism to the level where the achieved speedup corresponds the number of functional units in a processor.

Keywords: PRAM, VLIW, multithreading

---

1. Introduction

Efficient parallel computers are hard to manufacture due to yet unsolved theoretical and technical problems. The most important is: how to arrange efficient communication between processors?

A theoretically beautiful solution is to arrange communication by building a shared memory between processors [Schwarz66, Karp69, Fortune78]. In the 50's, 60's and 70's , however, the idea of building true shared memories was considered impossible. Instead several other possibilities like memory interleaving and memory distribution were investigated [Burnett70, Enslow74, Schwarz80].

Despite of their poor feasibility, the use of shared memory models continued among algorithm writers due to the simplicity of programming. The most widely used model is a Parallel Random Access Machine (PRAM) [Fortune78, Leighton91, McColl92] (see Section 2 for definition).

While building a shared memory has remained unfeasible, there has been a considerable effort to simulate a shared memory machine like PRAM with a distributed memory machine, which consists of processor and memory modules connected to each others through a communication network [Schwartz80, Gottlieb83, Gajski83, Hillis85, Pfister85, Ranade87, Abolhas91, Forsell96a].

One of the key components of such simulations is multithreaded processor. In this paper we outline a MultiThreaded VLIW processor Architecture with functional unit Chaining (MTAC), which is specially designed for efficient PRAM simulation.

MTAC features very short clock cycle, chained operations and absence of pipeline and memory system hazards. A preliminary performance evaluation of MTAC as a processor architecture is given. According to our experiments MTAC offers remarkably better performance than a basic pipelined RISC architecture and chaining improves the efficiency of instruction level parallelism to the level where the achieved speedup corresponds the number of functional units in processors.

2. The idea of simulation

In this section we outline briefly how a shared memory machine (SMM) like PRAM can be simulated on a distributed memory machine (DMM).

2.1 Parallel random access machine

The *Parallel Random Access Machine* (PRAM) model is a logical extension of the Random Access Machine (RAM) model. It is a widely used abstract model of parallel computation [Fortune78, Leighton91, McColl92].

A PRAM consists of P processors each having a similar register architecture. All processors are connected to a shared memory (see Figure 1).

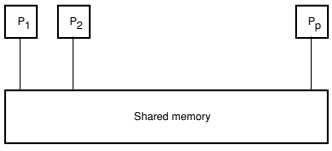

Figure 1. The PRAM model. $P_1,...,P_P$ are processors.

All processors run the same program synchronously, but a processor may branch within the program independently of other processors. That is, each processor has its own Program Counter.

The shared memory of large PRAMs is very difficult to build directly due to extreme complexity and very high costs: According to our investigations the silicon area and cost of a P-port shared memory are P^2 times greater than the required silicon area and cost of an ordinary singleport memory [Forsell94].

2.2 Distributed memory machine

Distributed memory machine (DMM) is a computer, which consists of processors and memory modules connected to each others through a communication network (see Figure 2). Processors use message passing to access memory locations.

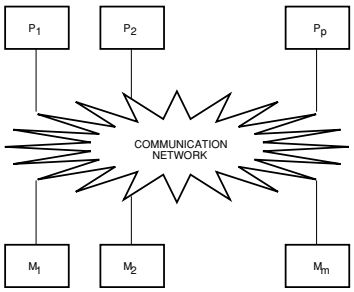

Figure 2. A distributed memory machine, where processors $P_1,...,P_p$ are connected to memory modules $M_1,...,M_m$ through a communication network.

Most current parallel computers are DMMs, because a DMM is a lot easier to build than a SMM [Hillis85, Prechelt93].

2.3 Simulating SMM on DMM

According to investigations DMMs can be used to simulate a SMM like PRAM [Schwartz80, Gottlieb83, Gajski83, Hillis85, Pfister85, Ranade87, Abolhas91, Forsell96a].

The standard solution uses two special techniques—random hashing and processor overloading.

Random hashing distributes memory locations to modules: A memory mapping function is randomly picked up from a family of memory mapping functions, which distribute memory locations evenly over the memory modules. The main job of random hashing is to prevent contention of messages in the communication network.

Processor overloading means simulating a number of virtual processors in a single processor. Overloading is used to hide the latency of long operations slowing down the execution of parallel programs: While e.g. a memory request spawned by an instruction of a virtual processor is being processed in a network, the processor executes instructions of the other virtual processors. This is possible because the instructions of virtual processors are independent of each others within a cycle of a physical processor.

The current RISC and superscalar processors with large register files and long reorder buffers and pipelines are not suitable for processor overloading, because they lack mechanisms for fast context switches. The objective of this paper is to outline a processor architecture that is capable for processor overloading and extracting enough instruction level parallelism.

3. Multithreading and chaining

In this section we describe two main techniques—multithreading and functional unit chaining—that are needed for an efficient PRAM processor architecture.

3.1 Multithreading

A processor is *multithreaded* if it is able to execute multiple processes (threads) of a single program in overlapped manner [Moore96].

The threads in a multithreaded processor are executed in fixed order or scheduled by a scheduling mechanism like priority or FIFO queue. The use of a complex scheduling mechanism usually requires that more than one instruction of a thread must be executed before next thread can be changed to execution. This is caused by the time taken by the scheduling decision [Moore96].

3.2 Functional unit chaining

In a processor containing multiple functional units one can chain the units so that an unit is able to use the results of its predecessors in the chain. We call this technique *functional unit chaining* (see Figure 3).

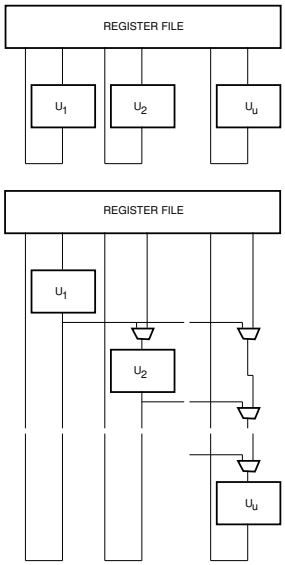

Figure 3. A set of functional units that operate in parallel and in chain.

Chaining allows execution of a code fraction containing dependencies within a single clock cycle of a thread. Unfortunately this significantly increases the length of the clock cycle.

Chaining can, however, be combined with superpipelining to retain short clock cycles. This solution generates dependencies in sequential code, but in the case of parallel algorithms, threads are independent within a cycle of a physical processor by definition.

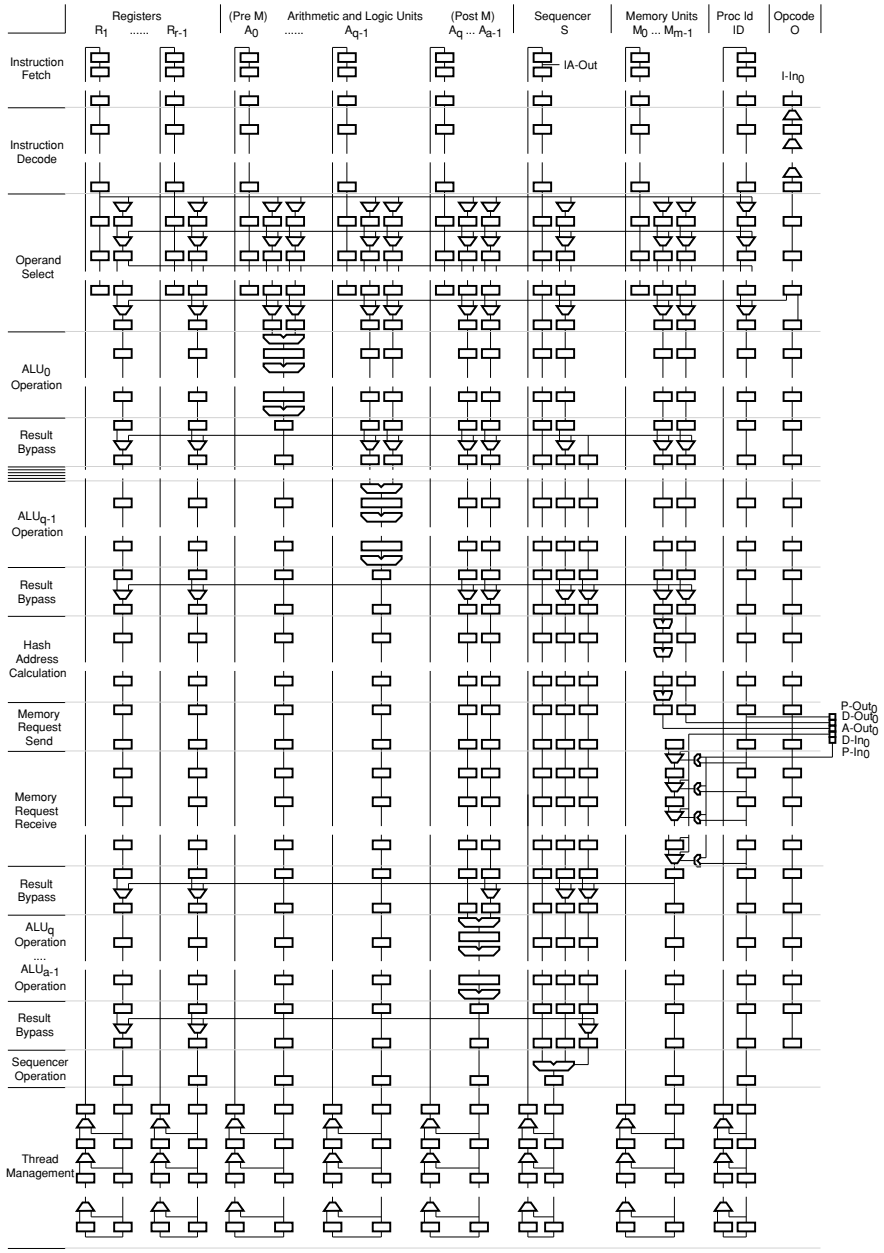

Figure 4. A detailed block diagram of a MTAC processor.

4. MultiThreaded Architecture with Chaining

In order to outline an abstract multithreaded processor architecture for PRAM simulation we apply functional unit chaining and fixed order multithreading to the basic VLIW core [Forsell96b]. We call the obtained architecture as the *MultiThreaded Architecture with Chaining* (MTAC).

The main parts of a MTAC processor are *a* arithmetic and logic units (A_0-A_{a-1}), *m* memory units (M_0-M_{m-1}), operation register (O), process identification register (ID), *r* general registers (R_0-R_{r-1}) and sequencer (S). Each part consists of a set of registers and processing elements connected to each others in chain-like manner. Some subparts of chains are also connected to each others (see Figure 4).

A vertical line of latches form a single thread (see Figure 4). A MTAC pipeline can store simultaneously $t/2$ to t threads. In every clock cycle threads are forwarded one step in the pipeline. Thus, the processor contains a variable number of threads which are cycling around the pipeline.

4.1 MTAC pipeline

The pipeline of MTAC consists of i instruction fetch stages (IF_0-IF_{i-1}), d decode stages (DE_0-DE_{d-1}), o operand select stages (OS_0-OS_{o-1}), e execute stages (EX_0-EX_{e-1}), h hash calculate stages (HA_0-HA_{h-1}), u memory request stages (ME_0-ME_{u-1}), b result bypass stages (BB_0-BB_{b-1}), one sequencer stage (SE) and 1-$t/2$ thread management stages (TM_0-$TM_{t/2-1}$). Symbols i, d, o, e, h, u, b and t are implementation dependent constants.

The cycle of a thread begins with the fetch of a very long operation code word form the local memory (IF stages). The latency of the memory is embedded into a i stages. During the DE stages the operation code of the instruction is decoded. Due to very short clock cycle d stages are needed for decoding. OS stages take care of operand selection and transfer to operand registers of appropriate functional units. To transfer all operands o stages are needed.

During the EX stages ALUs commit their operations. Due to varying order of instructions in typical basic block, first q ALUs (A_0-A_{q-1}) are placed before memory requests and the rest a-q ALUs (A_q-A_{a-1}) are placed after memory requests. Total e stages are needed for all execute operations.

In the middle of the EX stages physical addresses of memory references are calculated (HA stages) and all reference messages are transmitted simultaneously to the network (ME stages). Before the end of ME stages a thread receives the possible reply message of its memory request. Thus a latency of a memory request may be at most u clock cycles.

The processing of a thread ends with the selection of next PC address according to results of compare operations (SE stage). The last 1-$t/2$ stages are used for transferring the contents of threads to the beginning of the chain and allowing the number of threads to vary from $t/2$ to t.

4.2 Other aspects

We selected not to chain memory units, because sequential memory requests within an instruction cycle of a thread in multithreaded processor cannot be done without losing memory consistency or performing full synchronization between requests.

The order of the functional units is chosen according to typical instruction order in a basic block of code: Memory units are placed in the middle of ALUs. The sequencer is the final unit in the chain.

An ALU or sequencer may use the result of a compare operation committed in one of the previous ALUs to select one of its operands.

5. Performance evaluation

We evaluated the performance, static size of code and utilization of functional units for five processors—DLX, T5, T8, T11, T19 (see Figure 5).

	DLX	T5	T7	T11	T19
Functional units	4	4	6	10	18
ALU	1	1	3	6	12
CMP	1	1	1	1	1
MU	1	1	1	2	4
SEQ	1	1	1	1	1

Figure 5. Evaluated processors.

DLX is an experimental load/store RISC architecture featuring basic five level pipeline [Hennessy90]. It is included for comparison purposes as a representative of basic pipelined processor. T5, T7, T11 and T19 are instances of MTAC containing four,

six, ten and eighteen functional units plus one register unit respectively.

The numbers are chosen so that DLX and T5 have approximately the same amount of processing resources. The final ALU was changed to a compare unit (CMP) in MTAC processors for comparison purposes, because DLX has a dedicated hardware (CMP) for calculating and solving branch target addresses.

To measure exploited instruction level parallelism we included also T7, T11 and T19. T11 has twice the processing resources of T7 and T19 has two times the processing resources of T11. T5 was not used as a base machine for instruction level parallelism measurements, because it does have enough arithmetic power.

5.1 Simulation methods

The evaluation was made by simulating a single thread of execution of seven hand compiled toy benchmarks in the processors (see Figure 6).

PROG	NOTES
add	A program that calculates the sum of two matrixes
block	A program that moves a block of words to other location in memory
fib	A non-recursive program that calculates a fibonacci value
max	A program that finds the maximum value
pre	A program that calculates the presum of a matrix of words
sort	A program that sorts a table of words using recursive mergesort algorithm
trans	A program that returns the transpose of a matrix of words

Figure 6. The benchmark programs.

These benchmarks represent much used primitives of parallel programs. They were chosen for simplicity, because we wanted to use hand compilation to make sure we were measuring the performance of architectures not compilers.

The benchmarks were run assuming processors have ideal distributed memory system, i.e., every memory request is completed before the result is used. We assumed also that all processors were able to run at the same clock frequency and DLX is able to run multithreaded code perfectly. Without these two final assumptions DLX would have performed more than 100 times worse.

5.2 Results

We measured the execution time in clock cycles, static program code size in instructions and utilization of functional units for all processors. The normalized results are shown in Figure 7, Figure 8 and Figure 9 respectively.

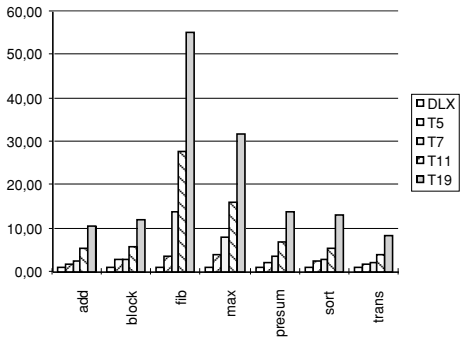

Figure 7. The relative performances of the evaluated processors.

According to the tests T5 runs benchmarks 2.7 times faster than DLX. T7, T11 and T19 perform 4.1, 8.1 and 16.6 times better than DLX.

The exploited instruction level parallelism in MTAC seemed to scale up exceptionally well in this case of small number of functional units: T11 was 1.97 times faster than T7 and T19 was 2.05 times faster than T11.

The static code size reduces almost as one might expect by execution time measurements. According to measurements the code size in instructions of T5, T11 and

T19 are 0.38, 0.28, 0.22 and 0.20 times the code size of DLX respectively.

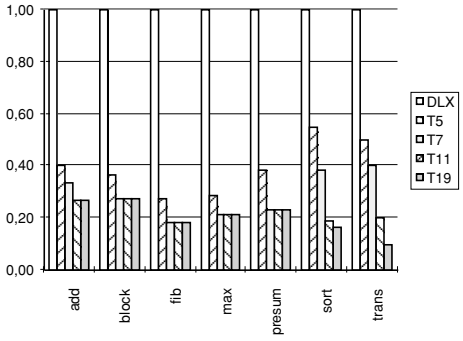

Figure 8. The relative static size of code of the evaluated processors in instructions.

The size of code would have been reduced more in the case of longer basic blocks, because in some of the benchmarks main blocks reduced into a single instruction that is not compressible.

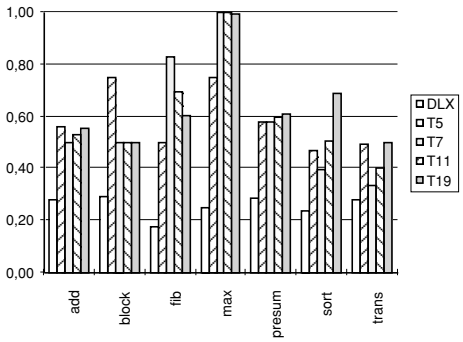

Figure 9. The utilization of the functional units in the evaluated processors.

The average utilization of functional units in DLX was only 25.8%. In the T5, T7, T11 and T19 average utilizations were 58.6%, 59.2%, 60.4% and 63.5% respectively. Thus, the utilization in MTAC processor was more then twice the utilization in DLX.

6. Feasibility

Currently many research and development projects going on in the area of multithreading [Moore96]. Among them Tera Computer Corporation is planning to release TERA MTA supercomputer before the end of 1996. TERA MTA runs at 333 MHz and it is capable of using up to 128 threads per processor [Alverson90]. TERA MTA does not, however, have distributed register file architecture of MTAC and fails to provide the instruction level parallelism and degree of superpipelining of MTAC.

MTAC processors contain a large number of components compared to basic pipelined processors like DLX. This increases the silicon die area and manufacturing costs of MTAC.

On the other hand, DLX is cannot compete with MTAC in the area of parallel computing, because MTAC is superior in dealing with processor overloading: MTAC switches thread in every clock cycle, whereas DLX uses at least 70 clock cycles to switch to other thread. In addition to that, MTAC features multithread pipeline and it can be extensively superpipelined without a fear of dependencies whereas DLX features single threaded execution with five level single thread pipelining.

A clear limitation of MTAC is that the executed program should have about twice the memory latency virtual processors per single MTAC processor. This limits the use of MTAC into the cases, where there is much parallelism in the algorithm.

7. Conclusions

We have described techniques to implement efficient processor overloading for PRAM simulations. We applied these techniques to a basic VLIW architecture. As a result we outlined the MultiThreaded Architecture with Chaining (MTAC).

We evaluated four MTAC processors and one reference processor by simulations. The results are favorable for MTAC:

T5—a five unit implementation of MTAC—performs 2.7 times better than a basic pipelined processor with the same processing resources. T7, T11 and T19—six, ten and eighteen unit implementations of MTAC—perform respectively 4.1, 8.1 and 16.6 times better than the basic pipelined processor.

The better performance comes from better exploitation of available instruction level parallelism, which is due to more than two times better utilization of functional units, absence of pipeline and memory system hazards, and the ability of MTAC to execute code blocks containing true dependencies within a single clock cycle.

In addition to better performance, the absence of pipeline and memory system hazards implies that there is no need for cache memories to get full performance and everything in MTAC can be extensively pipelined, which allows for a very short clock cycle.

Chaining seems to improve the exploitation of instruction level parallelism to the level where the achieved speedup corresponds to the number of functional units in a processor at least in the case of small number of units: T11 which has almost twice the processing resources of T7 performs two times better than T7 and T19 which has almost two times the processing resources of T11 performs two times better than T11.

Chaining also simplifies the compiler technology required to achieve full performance, because MTAC can execute code containing true dependencies.

A clear limitation of MTAC is that the executed program should have about twice the memory latency virtual processors per single MTAC processor. This limits the use of MTAC into the cases, where there is much parallelism in the algorithm. On the other hand, problems with limited parallelism are usually small enough to be executed in serial machine.

References

[Abolhas91]
F. Abolhassan, J. Keller and W. J. Paul, On the Cost-effectiveness and Realization of the Theoretical PRAM Model, FB 14 Informatik, *SFB-Report 09/1991*, Universität des Saarlandes, Saarbrucken, 1991.

[Alverson90]
R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield and B. Smith, The Tera Computer System, *Proceedings of the International Conference on Supercomputing*, June 11-15, 1990, Amsterdam, The Netherlands, 1-6.

[Burnett70]
G. J. Burnett and E. G. Coffman, A Study of Interleaved Memory Systems, *AFIPS Conference Proceedings SJCC*, 36 (1970), 467-474.

[Enslow74]
P. H. Enslow, *Multiprocessors and parallel processing*, John Wiley&Sons, New York, 1974.

[Feldman92]
Y. Feldman, E. Shapiro, Spatial Machines: A More Realistic Approach to Parallel Computation, *Communications of the ACM* **35**, 10 (1992) 61-73.

[Forsell94]
M. Forsell, Are multiport memories physically feasible?, *Computer Architecture News* **22**, 4 (September 1994), 47-54.

[Forsell96a]
M. Forsell, V. Leppänen and M. Penttonen, Efficient Two-Level Mesh based Simulation of PRAMs, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, June 12-14, 1996, Beijing, China, 29-35.

[Forsell96b]
M. Forsell, Minimal Pipeline Architecture—an Alternative to Superscalar Architecture, *Microprocessors and Microsystems* **20**, 5 (1996), 277-284.

[Fortune78]
S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proceedings of 10th ACM STOC* (1978), 114-118.

[Gajski83]
D. Gajski, D. Kuck, D. Lawrie and A. Sameh, CEDAR-A Large Scale Multiprocessor, *Proceedings of International Conference on Parallel Processing* (1983), 524-529.

[Gottlieb83]
A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing a MIMD, shared-memory parallel machine, *IEEE Transactions on Computers* **C-32**, (1983) 175-189.

[Hennessy90]
J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

[Hillis85]
W. D. Hillis, The Connection Machine, *The MIT Press*, 1985.

[Karp69]
R. M. Karp and R. E. Miller, Parallel Program Schemata, *Journal of Computer and System Sciences* **3**, 2 (1969) 147-195.

[Leighton91]
T. F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, 1992.

[McColl92]
W. F. McColl, General Purpose Parallel Computing, *Lectures on Parallel Computation Proceedings 1991 ALCOM Spring School on Parallel Computation (Editors: A. M. Gibbons and P. Spirakis)*, Cambridge University Press, Cambridge, 1992, pp. 333-387.

[Moore96]
S. Moore, *Multithreaded Processor Design*, Kluwer Academic Publishers, 1996.

[Pfister85]
G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E.A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of International Conference on Parallel Processing* (1985), 764-771.

[Prechelt93]
L. Prechelt, Measurements of MasPar MP-1216A Communication Operations, *Technical Report 01/93*, Universität Karlsruhe, Karlsruhe, 1993.

[Ranade87]
A. G. Ranade, S. N. Bhatt, S. L. Johnson, The Fluent Abstract Machine, *Technical Report Series BA87-3*, Thinking Machines Corporation, 1987.

[Schwarz66]
J. T. Schwarz, Large Parallel Computers, *Journal of the ACM* **13**, 1 (1966) 25-32.

[Schwarz80]
J. T. Schwarz, Ultracomputers, *ACM Transactions on Programming Languages and Systems* **2**, 4 (1980) 484-521.