Flexible Fibering Scheme for Thick Control Flow Processors

Martti Forsell
Efficient Computing and Communications
VTT
Oulu, Finland

Abstract—*One of the latest abstractions of parallel computing is to join the threads flowing through the same path to entities called thick control flows (TCF). In them, components are called fibers to distinguish them from threads having individual control. The abstraction differs from ordinary vector/SIMD computing by allowing a programmer to change the thickness dynamically during execution and guaranteeing that fibers are executed synchronously. While this has been shown to simplify programming, save silicon area of implementations and reduce energy consumption in execution, the first processor architecture optimized for TCF execution uses a blocking function for distributing the fibers evenly to the execution units and supporting multioperations that are special primitives of parallel computing implementing reductions. Unfortunately this implies that the on-core state of computation gets misaligned whenever the thickness is changed. In this paper we propose a flexible fibering scheme for thick control flow processors in which a programmer can dynamically select a suitable fibering function from a family of predefined functions. In our evaluation we show that by just by adding the interleave function the performance of test algorithms can be improved substantially and the misalignment problem is removed in many cases. Programming examples are given.*

Keywords—Parallel computing, processor architecture, TCF, fibering, programming

1 Introduction

The Flynn's taxonomy [Flynn72] classifies parallelism in computer architectures at high level according to number of instruction and data streams. The main alternatives are the sequential *single instruction stream single data stream* (SISD), data parallel *single instruction stream multiple data streams* (SIMD), as well as control and data parallel *multiple instruction stream multiple data stream* (MIMD) organizations. It is well-known that the parallel organizations are much faster than the SISD organization in most computations (see Figure 1 for a functionality of 36 operations executed in a SISD architecture as well as SIMD and MIMD architecture having four execution units). The difference between the parallel organizations is that SIMD offers more cost-efficient processing than MIMD in homogeneous computations while MIMD is faster if the executed functionality contains control parallelism or heterogeneity (see Figure 1). Our in-between model [Leppänen11] joins the threads flowing through the same path to entities called *thick control flows* (TCF) for keeping the simplicity and low cost-efficient implementability of SIMD but avoiding its strict homogeneity requirements by allowing multiple TCFs to be executed in parallel/overlapped way. The components of TCFs are called *fibers* to distinguish them from threads having individual control. The abstraction differs from SIMD by letting a programmer to change the *thickness* (the number of fibers) during execution and guaranteeing that fibers are executed synchronously. Usage of the TCF model has been shown to simplify programming, save silicon area of implementations and reduce energy consumption in execution [Forsell16, Forsell17]. Figure 1 shows how the partially heterogeneous functionality of 36 operations is executed in TCF model faster than in SIMD.

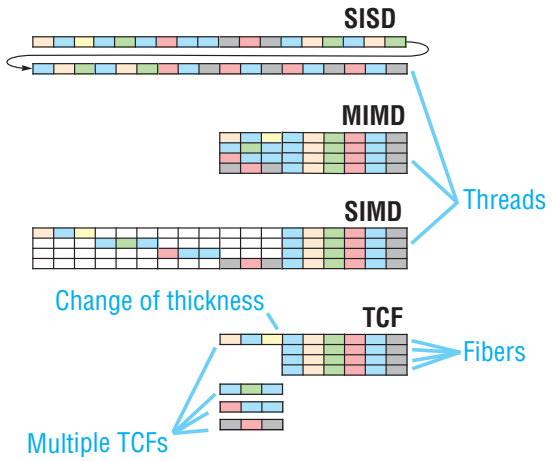

Figure 1. SIMD, MIMD and SIMD computations of partially heterogeneous functionality of 36 operations (the color reflects different operation). The TCF computation of the same functionality with 4 TCFs executed in parallel/overlapped way.

The first processor architecture optimized for TCF execution uses a blocking function for distributing the fibers evenly to the execution units and supporting multioperations that are special primitives of parallel computing implementing reductions [Forsell16]. Unfortunately, this implies that the on-core state of computation gets misaligned whenever the thickness exceeding the number of processing units is changed.

In this paper we propose a flexible fibering scheme for thick control flow processors in which a programmer can dynamically select a suitable fibering function from a family of predefined functions. In our evaluation we show that by just by adding the interleave function the performance of test algorithms can be improved substantially and the

ISBN: 1-60132-487-1, CSREA Press ©

misalignment problem is removed in many cases. Examples of applying the proposed scheme to simple algorithms are given.

The rest of this paper is organized as follows: In Section 2 we take a look at TCF-aware processor architectures, in Section 3 we propose a flexible threading scheme for TCF-aware processors, a preliminary evaluation of the proposed technique is given in Section 4, and finally in Section 5 we give our conclusions.

2 Thick control flow processor architectures

From a computational point of view, thick control flows are flexible data parallel abstractions of threads. Executing them efficiently requires a generalization of a multithreaded execution architecture. In this section, we have a look at TCF-aware architectures with a help of our TPA architecture that implements the TCF model on a top of advanced shared memory emulation architecture [Ranade91, Forsell14].

The *Thick Control Flow Processor Architecture* (TPA) [Forsell16] is the first TCF-aware processor architecture. Structurally, a TPA hardware is divided to frontend units, processing the TCF-level control and common parts, and backend units taking care of individual fibers (see Figure 2). The frontends (FE) are connected to a *Non-Uniform Memory Access* (NUMA) memory system and TCF buffers while the backend (BE) units are connected to replicated register blocks. TCF buffers store the data of active TCFs and replicated register blocks fiberwise data. Since TPA supports unbounded number of fibers replicated register blocks have an overflow mechanism to store the data not fitting to on-chip register block to external memory system.

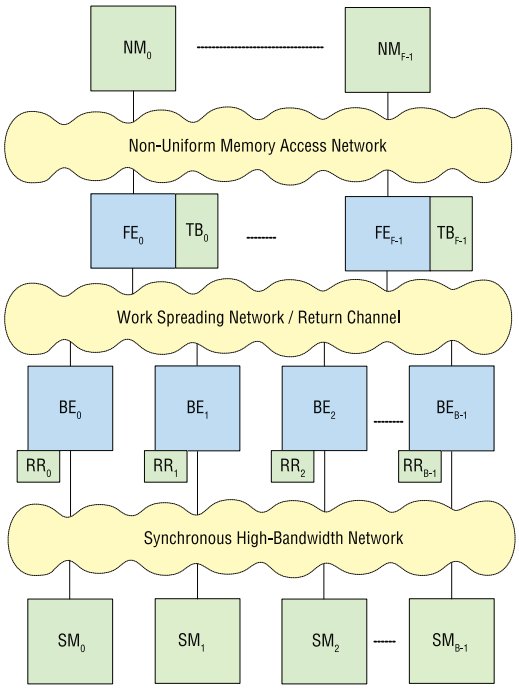

Figure 2. The high-level structure of TPA processor. FE=frontend unit, NM=NUMA memory module, TB=TCF buffer, BE=backend unit, RR=replicated register block and SM=shared memory module.

TPA starts a new instruction either by continuing execution of the current TCF or by switching to the next TCF if the previous instruction contains the STCF subinstruction. The instruction data pointed by the PC is then fetched to the frontend and executed like in an ordinary SMP/NUMA processor. If the instruction contains the PAR subinstruction, the FE selects operands and sends them to BEs through the work spreading network. A BE waits until a TCF with BE subinstructions is received from the work spreading network. Then the BE generates fibers to the BE pipeline according to the thickness information and executes them in it in overlapped way. After the last fiber has been generated, the BE checks again if there are new incoming TCFs. As the last fiber achieves the end of the pipeline the backend optionally send a return value to the frontend. Executing a single instruction in the frontend and backends is called a step of execution.

Like emulated memory architectures, TPA employs latency hiding via multifibering, extremely low-cost synchronization and exploitation of virtual instruction level parallelism via chaining of functional units [Ranade91, Forsell14]. Hot spots in intercommunication are minimized by using hashing of shared memory locations.

3 Programmable fibering scheme

The duty of the fibering scheme is mapping the fibers of TCF execution to the backends so that the properties of TCF-aware processors are realized. The fibering scheme used in the baseline *F*-frontend and *B*-backend TPA executing a TCF with thickness T maps the fibers to backend units so that fiber f, $0 \le f < T$ is executed in backend $f \text{ div } B$. There is no particular reason for this selection except that $f \text{ div } B$ is easy to calculate if B is a power of two, it is proven to work well in many cases, and multiprefix operations require that fibers are processed in increasing order in each backend. Despite of these good properties it also has a weakness of moving fibers between backends in the case of thickness change for $T > B$. Figure 3 shows execution of a logarithmic algorithm processing an array of N data elements so that the thickness reduces with exponentially increasing reductions from N-1 down to $N/2$. As the algorithm proceeds from an iteration to

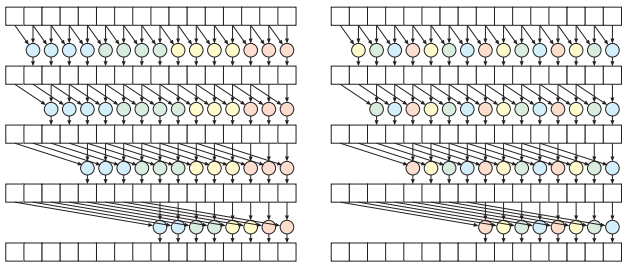

Figure 3. The blocked $f \text{ div } B$ and reverse interleaved $T - f \text{ mod } B - 1$ fibering of logarithmic algorithm in which thickness T drops from N-1 to $N/2$ for the case of N=16 and B=4. The rectangles represent data elements and circles represent arithmetic operations. Colors of the operations indicate the backend in which the fibers and operations are executed.

another, we observe the alignment problem where the fibers numbered from left to right move gradually a backend to another.

To remove these limitations, we propose replacing the fixed fibering scheme with a programmable one in which a programmer can select a fibering function from a family of preselected functions. In particular, we want to include the interleaving function mapping fiber f to backend $f \bmod B$ that does not move fibers between backends and therefore makes optimizations relying of fiber-wise context, i.e., replicated registers untouched. While the logarithmic algorithm of Figure 3 does not directly benefit from interleaved fibering since the starting element moves from left to right, it is easy to solve this issue by reversing the ordering of fibers from right to left. This reversed interleaved fibering, $T - f \bmod B - 1$, retains the alignment between the data elements and backends. Actually, the basic interleaved fibering can do the same thing if the data elements are just indexed in reverse order (see the realization of the multiprefix benchmark in Section 4.2).

An obvious side effect of changing the fibering function during execution is that the data stored to replicated register blocks does not move along with the fibers. This can sometimes implement useful functions for free. If the replicated register block contains intermediate results or data elements loaded from an array using the fiber identifier, switching from blocked fibering to interleaved fibering implements transpose-like row-column exchange between the fibers and data elements in replicated register block. Even though it is many times possible to implement the same functionality using row-column exchange memory access pattern in the same number of clock cycles, there are cases in which the number of functional units free from other duties is not high enough to implement the operations as fast as using the fibering change.

4 Evaluation

In order to evaluate the goodness of the proposed flexible fibering scheme for TCF-aware processors, we measure the performance of it on TPA, discuss how it influences programming as well as silicon area and power consumption of a hardware implementation.

4.1 Performance

To showcase performance effects of the flexible fibering scheme, we measured the execution time of four benchmark programs (see Table 1) on two TPA configurations (see Table 2).

To guarantee fair comparison and eliminate the effect of compiler, the test programs were written in TPA assembler and optimized by hand so that their performance is bounded only by the memory bandwidth. Execution of the benchmarks and running time measurements were done with our clock-accurate TPA simulator modified for flexible fibering technique.

The results of the measurements are shown in Figure 4. We can make the following observations from the measurements:

• Unlike the fixed fibering scheme of the baseline TPA using blocking function, the proposed flexible fibering scheme retains the alignment between data elements and backend units over changes of thickness if the interleaved fibering function is used.

prefix-b	A parallel program that calculates the prefix sum of an array of N integers using the logarithmic prefix sum algorithm and blocking fibering (see Figure 6). (*Executed in the baseline TPA.*)
prefix-i opt	A parallel program that calculates the prefix sum of an array of N integers using the logarithmic prefix sum algorithm optimized for interleaved fibering (see Figure 6). (*Executed in the TPA with flexible fibering scheme.*)
row-col-exchg-b	A parallel register to memory program that calculates a transpose-like row-column exchange of an array of $N=T_p \times B$ integers using blocking fibering. (*Executed in the baseline TPA.*)
row-col-exchg opt	A parallel register to memory program that calculates a transpose-like row-column exchange of an array of $N=T_p \times B$ integers using blocking to interleaving fibering change. (*Executed in the TPA with flexible fibering scheme.*)

Table 1. Benchmark programs in e language.

	TPA baseline	**TPA with flexible fibering scheme**
Number of FEs	1	1
Number of BEs	16	16
Number of FUs (FE/BE)	4/10	4/10
Fibering scheme	fixed (blocked)	flexible (blocked, interleaved)

Table 2. Tested architectures.

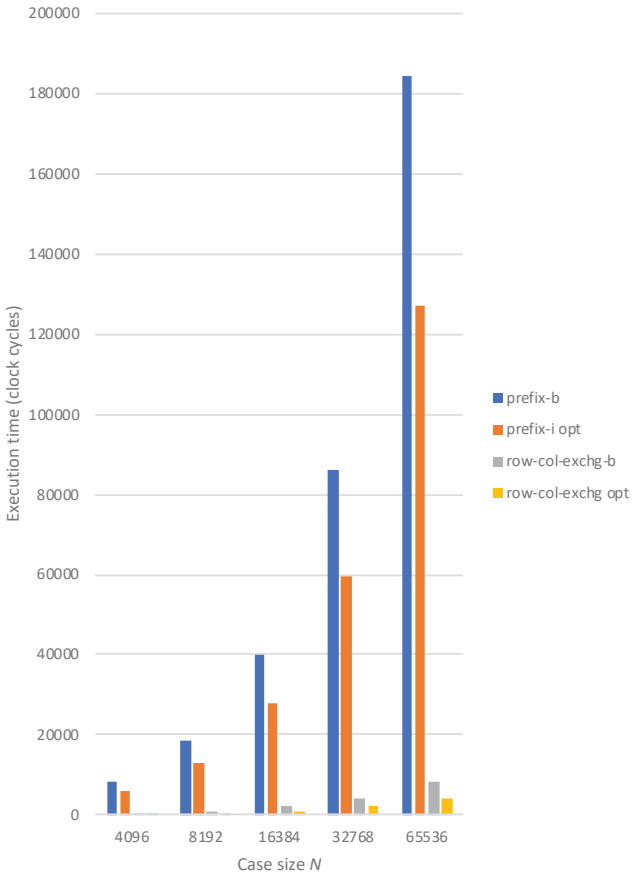

Figure 4. The execution time of the benchmark programs in the baseline TPA and TPA with the proposed flexible fibering scheme.

• Switching the fibering function does not take any time. It can be used to implement transposing-like row-column exchange operation for free if the data is already in replicated registers.
• With a help of the interleaved fibering, the execution time of the logarithmic prefix sum benchmark drops by up to 31%.
• Employing fibering function exchange drops the execution time of register to memory row-column exchange benchmark by 50%. If the number of functional units is increased in the baseline TPA, this performance gap however disappears.
• The execution time with the blocked fibering appears to be almost identical than that with interleaved fibering. This is enabled by the high-bandwidth interconnection network of TPA and hashing function designed to eliminate (most) hot spots in communication.

4.2 Programmability

In order to demonstrate effect of the proposed architectural technique on programmability, consider a prefix sum computation of an array of *N* integers with the logarithmic textbook algorithm. Although the algorithm is known suboptimal for the cases in which *N* exceeds the number of processing units, it is descriptive as an example since it makes use of a non-trivial communication pattern and iteration-wisely decreases the thickness from *N*-1 down to *N*/2 (see Figure 5). It also highlights the synchronous model of computation [Fortune78] used in TPA and simplicity of the TCF model. Unlike for the asynchronous model used in current multicores [Swan77, Lenoski92], there exists a well-founded theory of parallel algorithms for it [Jaja92].

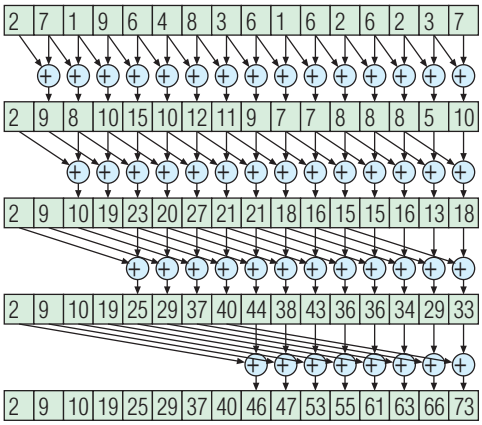

Figure 5. Operation of the logarithmic prefix sum algorithm for an array of 16 integers. The number of processors decreases from 15 to 8 along the run of the algorithm.

Let us now assume that the case size *N* is greater than the number of backend units *B*. For the TPA employing the TCF model there is no need to use loops for match the number of SW fibers to HW execution units. Since execution of TCFs is synchronous in TPA, there is no need for explicit barrier synchronizations or the temporary variables to emulate thick parallel computation on a narrow hardware. The code for a TPA is shown in Figure 6 as a high-level e language program, where the thickness of current TCF is denoted with **#** and the fiber identifier is denoted with **$** .

```
// Baseline TPA version (# goes down from N-1 to N/2, fibers 0..$-1 are assigned from A_[i] to A_[N-1] )
int A_[N];                          // Shared array of integers
int i;                              // Shared loop variable
for (i=1, #N-1; i<N; #-=i, i<<=1)
      A_[$+i] += A_[$];

; Blocked fibering version (1 instruction header, 3 instruction loop body) — prefix-b
    OP0   1      OP1   0     LSUB0 R33,O0 LWB1  O0    LWB3  O1     LWB33  LA0                                  ; Loop header

L0  OP0   _A_    OP1   3     OP2   1      OP3   176   LSUB0 R33,R3 LSHL1  R1,O2  TRAP  O3    LWB1  LA1   LWB2  R1    LWB33  LA0   SHL0  FID,O1 \
    SHL1  R2,O1  ADD2  A0,A1 ADD3  O0,A2  LDD0  A3    WB0   M0     RT1           STCF                    ; Load A_[$+i]

    OP0   _A_    OP1   3     OP2   65536  LSLT  R1,O2 RD0          SHL0  FID,O1  ADD1  O0,A0 LDD0  A1    ADD5  B0,M0 WB0    A5    RT1   STCF \
                                                                                                           ; Load A_[$]

    OP0   _A_    OP1   3     OP2   L0     BNEZ  O2    LWB3  R2     RD0           SHL0  FID,O1 SHL1  R2,O1 ADD2  A0,A1 ADD3  O0,A2 STD0  B0,A3 \
    RT1          STCF                                                                                      ; Store A_[$+i] + A_[$]
```

```
// Enhanced TPA version (# goes down from N-1 to N/2, fibers 0..$-1 are assigned from A_[N-1] down to A_[i] )
int A_[N];                          // Shared array of integers
int i;                              // Shared loop variable
for (i=1, #N-1, O(inter); i<N; #-=i, i<<=1){Pragma}
      A_[N-1-$] += A_[N-1-$-i];

; Interleaved fibering version (1 instruction header, 2 instruction loop body) — prefix-i opt
    OP1   3      OP2   1     OP3   176   OP4   0     LSUB0 R33,R3 LSHL1  R1,O2  TRAP  O3    LWB1  LA1   LWB2  R1    LWB3  O4    LWB5  R1    \
    LWB33 LA0    SHL0  FID,O1 SUB1 R4,A0  LDD0  A1    WB0   M0     RT1           INTER       STCF                    ; Loop header, load A_[N-1-$] to replicated reg.

L0  OP1   3      OP2   4096  LSUB0 R33,R3 LSLT  R1,O2 LWB2  R5     LWB33  LA0   RD0           SHL0  FID,O1 SUB1  R4,A0 SHL2  R2,O1 SUB3  A1,A2 \
    LDD0  A3     ADD5  B0,M0 WB0   A5     RT1         STCF                                                                ; Load A[N-1-$-i]

    OP1   3      OP2   L0    OP3   1      LSHL1 R1,O3 BNEZ  O2     LWB1   LA1   LWB3  R2    LWB5  R1    RD0           SHL0  FID,O1 SUB1  R4,A0 \
    STD0  B0,A1  RT1         STCF                                                                                        ; Store A_[N-1-$+i] + A_[N-1-$]
```

Figure 6. The logarithmic multiprefix addition as a high-level e-language programs and TPA assembler realization for the blocked fibering (prefix-b) and interleaved fibering (prefix-i opt). The INTER subinstruction switches to interleaved fibering from the default blocked fibering.

The program adjusts the thickness in the header of the **for**-loop from *N*-1 down to *N*/2. There is no need for synchronization due to the synchronous nature of the model. Finally, this program is leads to better utilization of the execution units than architectures with fixed number of threads because TCF does neither have the thread handling overheads nor quantization effects of the fixed threading schemes [Forsell16]. If the target machine dows not support flexible fibering (and interleaved fibering), the program compiles to a single instruction header and three instruction loop body (see Figure 6).

Employing interleaved fibering for this kind of situations may be provided by the compiler or instructed explicitly by the programmer (see Figure 6). The resulting TPA assembler version includes a single instruction loop header and two instruction loop body since one can keep the current element A_[*N*-1-$] in a replicated register and load only A_[*N*-1-$-*i*] if the indexing is done in the reverse order, while in the case of blocked fibering, one must load both A_[$+*i*] and A_[$] to the processor on each iteration.

4.3 Implementation considerations

Implementing the proposed flexible fibering scheme with a small number of fibering functions requires adding logic for calculating the functions and a subinstruction for specifying the used fibering. Since this builds on a top of already existing fibering logic units and instructions are fetched only every T/B clock cycle, the silicon area and power consumption overheads of flexible fibering is estimated to be very small with respect to the total area and power of the TPA chip multiprocessor [Forsell16].

5 Conclusions

We have proposed a flexible fibering scheme for TCF-aware processors. The idea is to let programmer select the mapping of fibers to backend units from a set of predefined functions. According to our experiments on a 16-backend TPA processor, just by adding interleaved function allows replicated registers to survive through the thickness changes. This speeds up execution of test algorithms reducing the thickness iteratively by up to 45-100% with respect to the TCF-aware processors with fixed threading scheme. In terms of the silicon area and power consumption, the proposed solutions does not have noticeable effect.

Since the speedups of the proposed fibering scheme appear highly application dependent, our future work includes more thorough evaluation of interesting fibering functions and their effect on a wider set of algorithms. We plan also to build a proof of concept prototype of TPA architecture and related methodology.

6 Acknowledgment

This work was funded by VTT and the grant 289773 of Academy of Finland.

7 References

[Flynn72] M. Flynn, Some Computer Organizations and their Effectiviness, *IEEE Transactions on Computers* **21**, 9 (1972), 948-960.

[Forsell14] M. Forsell and J. Roivainen, REPLICA T7-16-128 - A 2048-threaded 16-core 7-FU chained VLIW chip multiprocessor, *In the special session on Multicore, Manycore and Distributed systems at the 48th Asilomar Conference on Signals, Systems, and Computers*, November 2-5, 2014, Pacific Grove, USA, 1709-1713.

[Forsell16] M. Forsell, J. Roivainen and V. Leppänen, Outline of a Thick Control Flow Architecture, *Proc. 5th Workshop on Parallel Programming Models Special Edition on Task Parallelism*, October 26-28, 2016, Marina del Rey Marriott, Los Angeles, USA.

[Forsell17] M. Forsell, J. Roivainen, V. Leppänen and J. Träff, Supporting Concurrent Memory Access in TCF-aware Processor Architectures, *Proc. 2017 IEEE Nordic Circuits and Systems Conference (NORCAS'17)*, October 24-25, 2017, Linköping, Sweden.

[Fortune78] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proc. 10th Annual ACM symposium on Theory of computing (STOC'78)*, San Diego, California, USA — May 1-3, 1978, 114-118.

[Jaja92] J. Jaja. *Introduction to Parallel Algorithms.* Addison-Wesley, Reading, 1992.

[Lenoski92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta,J. Hennessy, M. Horowitz, and M. Lam, The Stanford Dash Multiprocessor, *IEEE Computer* **25**, (March 1992), 63-79.

[Leppänen11] V. Leppänen, M. Forsell and J-M. Mäkelä, Thick Control Flows: Introduction and Prospects, *Proc. 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, July 18-21, 2011, Las Vegas, USA, 540-546.

[Ranade91] A. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences* **42** 307–326, 1991.

[Swan77] R. Swan, S. Fuller and D. Siewiorek, Cm*—A Modular Multiprocessor, *Proc. NCC'77*, 645-655, 1977.