Journal of Universal Computer Science, vol. 3, no. 9 (1997), 1037-1055
submitted: 5/12/96, accepted: 16/7/97, appeared: 28/9/97 © Springer Pub. Co.

MTAC - A Multithreaded VLIW Architecture for PRAM Simulation

Martti Forsell
(University of Joensuu, Finland
mforsell@cs.joensuu.fi)

Abstract: The high latency of memory operations is a problem in both sequential and parallel computing. Multithreading is a technique, which can be used to eliminate the delays caused by the high latency. This happens by letting a processor to execute other processes (threads) while one process is waiting for the completion of a memory operation. In this paper we investigate the implementation of multithreading in the processor-level. As a result we outline and evaluate a MultiThreaded VLIW processor Architecture with functional unit Chaining (MTAC), which is specially designed for PRAM-style parallelism. According to our experiments MTAC offers remarkably better performance than a basic pipelined RISC architecture and chaining improves the exploitation of instruction level parallelism to a level where the achieved speedup corresponds to the number of functional units in a processor.
Key Words: PRAM, VLIW, multithreading, chaining
Category: C.1.2

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 elegant solution is to arrange communication by building a shared memory between processors [Schwarz 66, Karp 69, Fortune 78]. 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 [Burnett 70, Enslow (74), Schwarz 80].

Despite of their poor feasibility, the use of shared memory models continued among algorithm designers due to the simplicity of programming. The most widely used model is parallel random access machine (PRAM) [Fortune 78, Leighton (91), McColl (92)] (see [Section 2] for a definition).

While building of shared memory has remained unfeasible, there has been a considerable effort to simulate an ideal shared memory machine like a PRAM with a physically distributed memory machine, which consists of processors and memory modules connected to each others through a communication network [Schwartz 80, Gottlieb 83, Gajski 83, Hillis (85), Pfister 85, Ranade 87, Abolhassan 93, Forsell 96a].

Efficient processor architectures are not as hard to design as entire parallel computers, but there are still two major problems that reduce the performance of them: Typical thread-level parallel processor architectures suffer from low utilization of processors, because a large portion of time is wasted in waiting for memory requests to complete, and typical instruction-level parallel architectures suffer from delays caused by instruction dependencies, because typical sequential code contains a lot of dependencies.

In this paper we try to find a common solution to both of these problems by using multithreading to increase the utilization of processors and functional unit chaining to eliminate dependencies. With this in mind, we apply multithreading and chaining along with extensive superpipelining to a basic very long instruction word (VLIW) [Fisher 83, Nicolau 84] (see [Section 3] for a definition) architecture [Forsell 96b]. As a result we outline a MultiThreaded VLIW processor Architecture with functional unit Chaining (MTAC), which specially suits for efficient simulation of a PRAM.

MTAC features a 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 with the same number of functional units and chaining improves the efficiency of instruction level parallelism to a level where the achieved speedup corresponds to the number of functional units in processors.

2 The idea of simulation

In this section we outline briefly how an *ideal shared memory machine* (SMM), like a PRAM, can be simulated by a physically distributed memory machine.

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 [Fortune 78, Leighton (91), McColl (92)]. A PRAM consists of *p* processors each having a similar register architecture. All processors are connected to a shared memory [see Fig. 1]. 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. There are no memory access restrictions: If all *p* processors initiate a memory reference simultaneously, the memory system will complete all references in a single clock cycle.

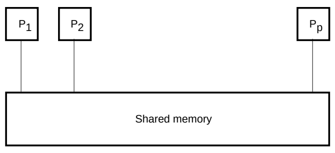

Figure 1: A Parallel Random Access Machine with processors $P_1,...,P_p$.

The ideal shared memory of large PRAMs is very difficult to build directly due to extreme complexity and very high costs: According to our earlier investigations multiport memory chips are feasible, but 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 single port memory [Forsell 94]. It is hard to imagine that multiport memories could be implemented by another structure that is simpler than that proposed in [Forsell94].

2.2 Distributed memory machine

A *physically distributed memory machine* (DMM) is a computer, which consists of *p* processors and *m* memory modules connected to each others through a communication network [see Fig. 2]. Processors use message passing to access memory locations.

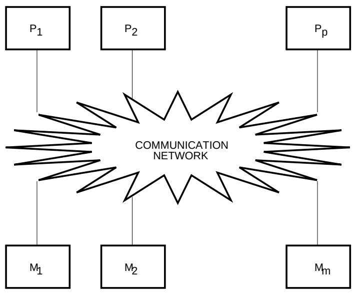

Figure 2: A physically distributed memory machine, where processors $P_1,...,P_p$ are connected to memory modules $M_1,...,M_m$ through a communication network.

Most current parallel computers are DMMs [Hillis (85), Prechelt 93], because a DMM is a lot easier to build than a SMM [Forsell 94].

Note that the term shared memory is currently widely used in the computer industry in a different meaning: A machine has a shared memory if all processors have access to a single memory, even if access is serialized. We consider this misleading because this terminology classifies both a parallel computer using the PRAM model and a parallel computer with a time-shared bus between processors and the memory to the same shared memory class, although the PRAM computer has considerably higher performance in most applications.

2.3 Simulating SMM on DMM

According to earlier investigations, DMMs can be used to simulate SMMs like PRAMs [Schwartz 80, Gottlieb 83, Gajski 83, Hillis (85), Pfister 85, Ranade 87, Abolhassan 93, Forsell 96a].

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

Random hashing is used to distribute memory locations of a SMM to the modules of a DMM: A memory mapping function is randomly picked up from a family of memory mapping functions, which distribute memory locations evenly over the memory modules. If the function does not work well with a particular application the function is changed to another. The main purpose of hashing is to prevent contention of messages in the communication network.

Processor overloading means simulating a number of virtual processors by a single physical processor. Processor overloading is used to hide the latency of long operations, which slow down the execution of parallel programs: While, e.g., a memory request initiated 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 [Johnson 89, Hennessy 90] are not suitable for processor overloading, because they lack mechanisms for fast process switches, and the number of registers is still far too small for efficient data prefetching. 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, distribution and chaining

In this section we describe four main techniques—multithreading, register file distribution, VLIW scheduling and functional unit chaining—that are necessary 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 an overlapped manner [Moore (96)]. The threads in a multithreaded processor are executed in fixed order or scheduled by a scheduling mechanism utilizing, e.g., a priority 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 [Moore (96)].

3.2 Register file distribution

An efficient realization of multithreading requires simultaneous access to a large number of registers, because the execution of multiple threads is overlapped in a multithreaded processor. This causes problems, because large multiport register files are slow and very difficult to build.

Quick simultaneous access to all registers can be implemented by a novel register file organization called *distributed register file*, in which single registers are treated as separate functional units that are connected to each other by a cross-bar like selection network where needed [see Fig. 4] [Forsell 96b].

3.3 VLIW scheduling

A *Very Long Instruction Word* (VLIW) processor is a processor, which executes instructions consisting of the fixed number of smaller subinstructions [Fisher 83, Nicolau 84]. Subinstructions are executed in multiple functional units in parallel. Subinstructions filling actual instructions are determined under compile time.

In order to achieve high speedups the programs for VLIW processors must be compiled using advanced compilation techniques breaking the basic block structure of the program [Fisher 81, Chang 91].

We selected the VLIW scheduling for MTAC, because the main alternative—the dynamic runtime scheduling with out of order execution and branch prediction used in superscalar processors [Johnson 89, Hennessy (90)]—is not suitable for strictly synchronous PRAM simulation. Furthermore, there is evidence that a VLIW processor with the same number of functional units is faster than a superscalar processor with out of order execution and dynamic branch prediction, while the VLIW solution requires no more silicon area [Forsell 96b].

3.4 Functional unit chaining

In a processor consisting of multiple functional units [Hennessy 90] one can chain the units so that a unit is able to use the results of its predecessors in the chain. We call this technique *functional unit chaining* [see Fig. 3].

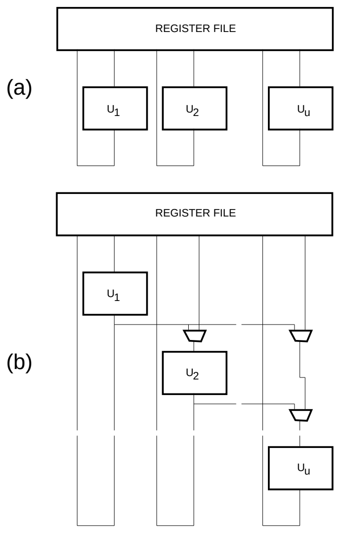

Figure 3: A set of functional units that operate (a) in parallel and (b) in chain. Symbols with two inputs and a single output are multiplexers.

3.5 Efficient parallel multithreading with chaining

Functional unit chaining allows for 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 [Jouppi 89] to retain short clock cycles. This solution generates delays due to dependencies in a sequential program code, but in the case of parallel programs, threads are independent within a cycle of a physical processor by definition.

4 Multithreaded architecture with chaining

In order to outline an abstract multithreaded processor architecture for a simulation of a PRAM we apply functional unit chaining, fixed order multithreading, and super-pipelining to the basic VLIW core with a distributed register file [Forsell 96b]. We call the obtained architecture the *MultiThreaded Architecture with Chaining* (MTAC).

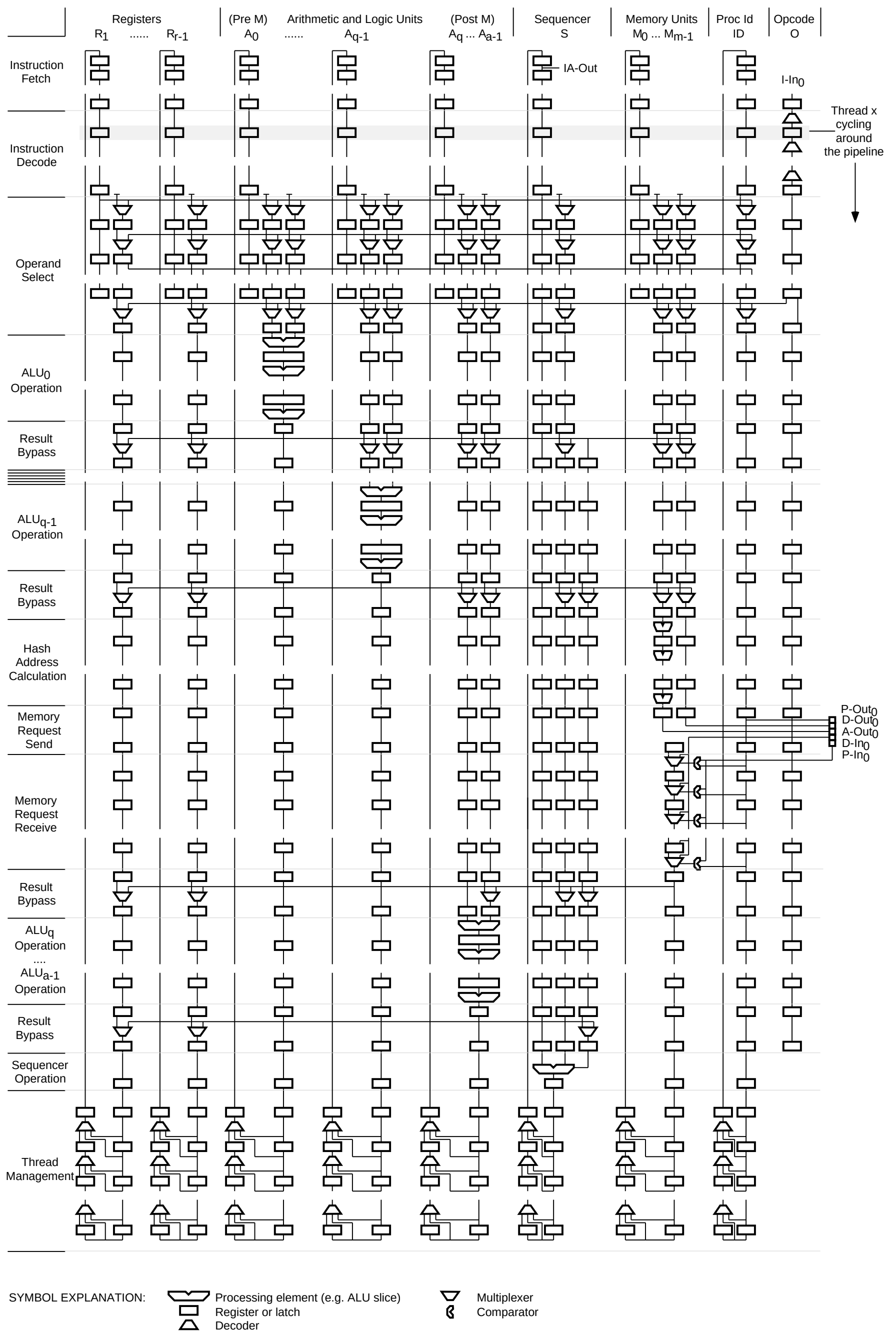

Figure 4: A detailed block diagram of a MTAC processor.

The main features of MTAC are

- Multithreaded operation with single instruction length threads and a zero-time switch between threads.
- Variable number of threads.
- Multiple functional units connected as a chain, so that a unit is able to use the results of its predecessors in the chain.
- A very short clock cycle due to a reqular structure without long lines that cannot be pipelined.
- VLIW-style instruction scheduling allowing for a complex coding of instructions if necessary.
- The construction of a compiler is easier than with average VLIW machines due to functional unit chaining allowing for the execution of dependent subinstructions within an instruction.
- No need for on-chip (or off-chip) data or instruction caches or branch prediction. Multithreading is used to hide memory, communication network and branch latencies.
- Completely queued operations — no memory reference message handling overhead.

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 a chain-like manner. Some subparts of chains are also connected to each others [see Fig. 4].

A horizontal line of latches store contents of a single thread [see Fig. 4]. From this point of view a MTAC processor consists of $t_{\max}$ slices, which store and process threads. Any of the lowermost $t_{\max}$-$t_{\min}$+1 slices can be conncted to the topmost slice so that the pipeline of MTAC can store simultaneously $t_{\min}$ to $t_{\max}$ 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 a 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 to $t_{\max}$-$t_{\min}$+1 thread management stages (TM_0-$TM_{t_{\max}-t_{\min}+1}$). Symbols *i*, *d*, *o*, *e*, *h*, *u*, *b*, $t_{\min}$, and $t_{\max}$ are implementation dependent constants.

The cycle of a thread begins with the fetch of a very long operation code word from the local memory (IF stages). The latency of the memory is embedded into *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 the operand selection and the transfer of operands 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 the varying order of instructions in a typical basic block, the first q ALUs (A_0-A_{q-1}) are placed before memory units and the rest a-q ALUs (A_q-A_{a-1}) are placed after memory units. Totally 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 communication network (ME stages). Before the end of ME stages a thread receives the possible reply messages of its memory requests. If a thread issuing a memory read request reaches the last ME stage without receiving the reply message, the whole processor is halted until the message arrives.

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

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 the 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. We added also a special mechanism to benefit more from chaining: 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.

Synchronization between processors and threads is be done by a separate synchronization network between processors: A processor is equipped with a mechanism, which freezes a thread after it has executed a synch instruction by blocking its memory references and PC changes until the processor receives a synch message from a separate synchronization network.

PRAM model assumes unrealisticly that the number of processors p can be as high as an algorithm requires, whereas the number of threads is limited to $t_{\max}$ in a MTAC processor. This is not, however, a difficult problem, because any task T_Q consisting of Q operations and taking U time steps with enough processors can be executed with

t processors in time $U+(Q-U)/t$ [Brent 74]. On the other hand, if the number of threads is set to $T < t_{\min}$, the processor creates $t_{\min}-T$ null threads, so that the processor remains functional. Alternatively, we can quite easily add a support for fast context switches to MTAC, if more than $t_{\max}$ simultaneous threads are absolutely required: Add w switch stages (SW) in the beginning of the pipeline of MTAC. During these stages a thread sends its contents to the local memory and receives the contents of a stored thread from the local memory.

5 Performance evaluation

We evaluated the performance, the static size of code and the utilization of functional units for five processors—DLX, T5, T7, T11, T19 [see Tab. 1].

UNITS	**DLX**	**T5**	**T7**	**T11**	**T19**
Functional Units	4	4	6	10	18
- Arithmetic and Logic Unit (ALU)	1	1	3	6	12
- Compare Unit (CMP)	1	1	1	1	1
- Memory Unit (MU)	1	1	1	2	4
- Sequencer (SEQ)	1	1	1	1	1
Register Unit	1	1	1	1	1

Table 1: Evaluated processors.

DLX is an experimental load/store RISC architecture featuring basic five level pipeline [Hennessy (90)]. It is included for comparison purposes as a representative of basic pipelined processor. DLX provides a good reference point for an architectural comparison, although there are faster sequential processors available.

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 the same number of functional units. 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 the exploited instruction level parallelism we included also T7, T11 and T19. T11 has twice the processing resources of T7 and T19 has twice 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 for proper comparisons.

We also evaluated the performance of six machines—D-1, D-16, D-16s, T5-1, T5-16 and T19-16 [see Tab. 2].

PROPERTIES	**D-1**	**D-16**	**D-16s**	**T5-1**	**T5-16**	**T19-16**
Type of processor	DLX	DLX	DLX	T5	T5	T19
Number of processors	1	16	16	1	16	16
Number of threads	1	512	1	512	512	512
Clock frequency (MHz)	300	300	300	600	600	600
Thread swithing time (clk)	-	70	-	0	0	0
Latency of a network (clk)	-	6	6	-	6	12
Memory module technology	IL	B	B	B	B	B
Interleaving factor	4	-	-	-	-	-
Number of memory modules	1	16	16	1	16	64
Number of banks in a module	-	32	32	64	64	64
Assumed number of bank collisions	-	3	3	3	3	3
Cycle time of a memory module (clk)	27	27	27	54	54	54
Access time of a memory module (clk)	15	15	15	30	30	30
Access time of a level 1 cache (clk)	1	-	-	-	-	-
Line length of a level 1 cache (word)	4	-	-	-	-	-
Cache miss time (clk)	16	-	-	-	-	-
Next cache line word available (clk)	4	-	-	-	-	-

Table 2: Evaluated machines. The numbers are assumptions based on available technology (see the discussion in [Section 6]). (IL =Interleaved, B=Banked).

D-1 is a sequential machine with a single DLX processor, a single cycle four-word line level 1 cache, and a 4-way interleaved DRAM memory system. In a case of a cache miss, we assume that, the cache line is filled so that first word is available after 16 clock cycles and the remaining are available after 4 cycle delay each. D-1 is included for comparison purposes as a representative of conventional sequential computer. D-1 provides a good reference point for a system level comparison, although there are faster sequential computers available.

D-16 and D-16s are parallel machines using sixteen DLX processors connected to each others with a communication network. D-16 uses 512 threads per processor in order to the hide message latency. D-16s uses a single thread per processor. T5-1 is a sequential machine with a single 512-thread T5 processor. T5-16, T19-16 are parallel machines using sixteen 512-thread T5 or T19 processors connected to each others with a communication network, respectively. The memory system of all parallel machines consists of 16 to 64 memory modules. A memory module is divided into 32 or 64 banks with access queues due to slow cycle time of a memory module.

5.1 Simulation methods

We made two kinds of experiments—processor level simulations and machine level estimations.

Processor level simulations were made by simulating a single thread of execution of seven hand compiled toy benchmarks [see Tab. 3] in the processors.

PROGRAM	**NOTES**
add	A program that calculates the sum of two matrices
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 of matrix of integers
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

Table 3: The benchmark programs.

Single thread simulations are based on the fact that in the most parallel applications the size of the problem, *N*, is much bigger than the number of the processors, *P*, in a parallel machine. Now, a typical way to spread the problem of size *N* to *P* processors is to divide the problem into *P* subproblems of size *N*/*P*. Then these *P* subproblems are solved with *P* processors using sequential algorithms. Finally, if necessary, the results of these *P* subproblems are combined with *P* processors. Usually the sequential part dominates the execution time. Now, it is possible to extract a single thread of execution from the sequencial part, and that is how these benchmarks were created. The purpose of these processor level simulations was to figure out the architectural characteristics of MTAC without the effect of a memory system.

The benchmarks, except fib, are frequently used primitives in 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 an ideal distributed memory system, i.e., every memory request is completed before the result is used. We assumed also that all processors are able to run at the same clock frequency and DLX is able to run multithreaded code perfectly.

Machine level estimations were made by simulating the execution of full versions of two hand compiled toy benchmarks (add and max) [see Tab. 3] in the machines and taking into account the effect of the assumed memory system. Due to randomized hashing of memory locations it is possible that two memory reference messages collide, i.e., they are trying to enter into a single memory bank simultaneously. In the case of collision, one message have to wait in an access queue until another has completed its memory access. We assumed that the maximum number of collisions is three. The purpose of these machine level estimations was to figure out the more realistic performance of a MTAC system. Due to a speculative nature of these estimations we included only two benchmarks.

We used *dlxsim* (version 1.1) [Hennessy (90)] for DLX programs and the *MTACSim* (version 1.1.0) simulator [Forsell 97] for MTAC programs. A code example for the max benchmark is shown in Table 4.

```
; An example code of max benchmark for T5-1 and T5-16 machines
;
; R1       Pointer to the table
; R2       The end address of the table
; R3       Memory(Ponter) contents
; R4       Max value
; R5       Thread interleave constant (threads * word/processor)
; R6       Thread number sifted for word/processor
; R7       Save address for combining
; R8       Combining pointer
; R9       Combining interleave stepping down
; R10      Constant 4

_MAIN

    SHL0   O0,O1  WB5    A0     OP0    THREAD        OP1    2
    SHL0   R30,O0 WB6    A0     OP0    2
    LD0    A0     ADD0   O0,R6  WB1    A0     WB4    M0     WB7    A0     OP0    _TABLE
    ADD0   R1,R5  WB1    A0     WB2    O0     WB10   O1     OP0    _ENDING       OP1    4

L0
    LD0    R1     WB3    M0     SLE    M0,R4  BNEZ   O1     OP1    L1
           ADD0   R1,R5  SLT    R1,R2  BNEZ   01,O0  WB1    A0     WB4    R3     OP0    L2     OP1    L0
L1  ADD0   R1,R5  SLT    A0,R2  BNEZ   01     WB1    A0     OP1    L0

L2  SHR0   R5,O0  WB9    A0     OP0    1
    ADD0   R7,R9  ST0    R4,R7  WB8    A0

L3
    SHR0   R9,O0  LD0    R8     WB3    M0     SLE    M0,R4  BNEZ   O1     WB9    A0     OP0    1      OP1    L4
           ADD0   R7,R9  ST0    R3,R7  SGE    R9,R10 BNEZ   O1,O0  WB4    R3     WB8    A0     OP0    L5     OP1    L3
L4  ADD0   R7,R9  SGE    R9,R10 BNEZ   O1     WB8    A0     OP1    L3

L5  TRAP   O0     OP0    0

_TABLE:
  .RANDOM         4096

_ENDING:
```

Table 4: An Example code of the max benchmark for T5-1 and T5-16 machines. Subinstructions written on a single line belong to the same instruction. See [Forsell 97] for further information.

5.2 Results

In our processor level simulations we measured the execution time in clock cycles, the static program code size in instructions and the utilization of functional units for all processors. The normalized results are shown in Figure 5.

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 faster 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, T7, T11 and T19 are 0.38, 0.28, 0.22 and 0.20 times the code size of DLX, respectively.

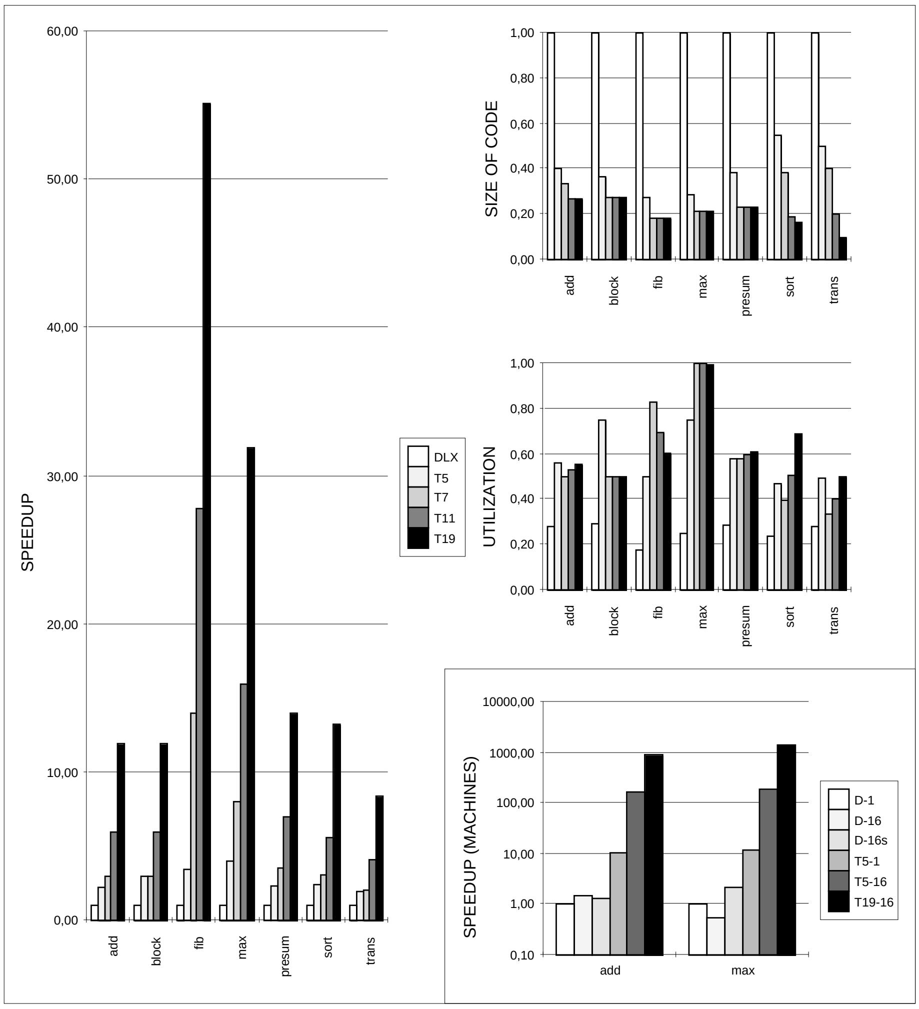

Figure 5: The results of experiments. The relative performances, the relative static code size and the utilization of functional units in the evaluated processors (upper left charts). The relative performances of the evaluated machines (the lower right chart).

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 was not compressible.

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 utilizations in MTAC processors were roughly twice than that in DLX.

In our machine level estimations we measured the execution time in milliseconds for all machines. The normalized results are shown in Figure 5.

According to the tests T5-1 runs add benchmark 10.5 times faster and max benchmark 12.0 times faster than D-1. T5-16 performs add 168, 109 and 126 times better than D-1, D-16 and D-16s. The numbers are 191, 354 and 88,5 for the max benchmark. T19-16 performs add 895, 583 and 671 times better than D-1, D-16 and D-16s. The numbers are 1460, 2700 and 675 for the max benchmark.

The performance of MTAC machines seemed to scale up exceptionally well in this case of small number of processors: T5-16 was 16.0 and 15.9 times faster than T5-1.

In the case of DLX systems, the execution time of benchmarks were capitalized by memory systems delays. The performance of these benchmarks would not be much higher even if the DLX processor of D-1 was replaced by a faster superscalar processor. The performance of parallel DLX machines, D-16 and D-16s, did not scale up from a single processor version, D-1. This due to inability of a DLX processor to deal with overloading.

6 Other projects and feasibility

Currently there are many research and development projects going on in the area of multithreading [Moore (96)]. Among them are *Saarbrücken Parallel Random Access Machine* (SB-PRAM) [Abolhassan 93] based on Fluent Abstract Machine [Ranade 87] and Berkley-RISC I [Patterson 82] style multithreaded processor [Keller 94], *Alewife* Project based on Sparcle processors in MIT [Agarwal 93], and *TERA MTA* supercomputer from Tera Computer Corporation [Alverson 90].

Unlike MTAC and TERA, SB-PRAM and Sparcle are not high speed designs: A SB-PRAM processor runs at 8 MHz and a MIT Sparcle processor runs at 33 MHz. The GaAs based TERA runs at 333 MHz and it is capable of using up to 128 threads per processor. However, a TERA does not have the a distributed register file architecture as a MTAC and fails to provide the instruction level parallelism and degree of superpipelining of MTAC.

MTAC processors contain a larger number of components compared to basic pipelined processors like DLX. This increases the silicon die area and manufacturing costs of MTAC processors. On the other hand, one does not need to use silicon area for large on-chip caches with MTAC saving die area for other use.

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

We believe that the clock frequency of a MTAC processor can be made as high as the clock frequency of fastest commercial processors (600 MHz) with current technology, or even higher, because there is no need for forwarding within a single clock cycle in MTAC, the structure of MTAC is very regular, and there are no long wires that can not be pipelined. In this sense MTAC closely resembles *Counterflow Pipeline Processor Architecture* (CFPP) from Sun Microsystems Laboratories and Oregon State University, which uses regular structure, local control and local communication to achieve maximum speed [Sproull 94, Janik 97]. CFPP is, however, not designed for parallel processing like MTAC.

We also believe that the maximum number of threads in MTAC $t_{\max}$ can be made as high as 500 with current technology, or even higher, because a MTAC processor can be divided to multiple chips placed in a multichip module, if it does not fit into a single chip. Thus, the latency of a fast communication network and the approximate 90 ns cycle time of current main memory building blocks, DRAM chips, could be hided in a parallel computer using MTAC processors and fast parallel communication system like CBM outlined in [Forsell 96a].

7 Conclusions

We have described techniques for reducing performance loss due to low utilization of functional units and delays caused by instruction dependencies in thread-level parallel and instruction-level parallel architectures. We applied these techniques along with extensive superpipelining to a basic VLIW architecture. As a result we outlined the MultiThreaded Architecture with Chaining (MTAC), which specially suits for efficient PRAM simulation.

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

T5—a five unit implementation of MTAC—performed 2.7 times faster than a basic pipelined processor with the same number of functional units. T7, T11 and T19—six, ten and eighteen unit implementations of MTAC—performed 4.1, 8.1 and 16.6 times faster than the basic pipelined processor, respectively.

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, the absence of pipeline and memory system hazards, and the ability of MTAC to execute code blocks containing 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. Furthermore, there are no long wires in MTAC, so everything in MTAC can be extensively superpipelined allowing for a very short clock cycle.

We also evaluated three machines based on MTAC procesors and three reference machines by simulations. The effect of of assumed memory systems was estimated and taken into account. The results are even better for MTAC based machines than in the case of plain processors:

T5-1—a machine with a single T5 processor and banked memory system—performs 11 times faster than a conventional sequential machine with a basic pipelined processor and a cached memory system. T5-16 and T19-16—16-processor machines based on T5 and T19 processors—perform respectively 90 to 350 and 580 to 2700 times faster than two 16-processor machines using the basic pipelined processor.

Functional unit 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 faster than T7 and T19, which has almost two times the processing resources of T11, performs two times faster than T11.

Chaining also simplifies the VLIW compiler technology required to achieve the full computing power, because MTAC can execute code containing true dependencies. Unfortunately the object code compiled for one MTAC processor is not compatible with another version of MTAC processor. This is a common problem with VLIW architectures, but it is a problem with superscalar processors too. The maximum performance cannot be achieved without recompilation.

A general purpose supercomputer cannot be based solely on MTAC, because there are certain strictly sequential problems that can be executed much faster with conventional processors. Moreover, high speed realtime applications may need faster response than MTAC can serve. A possible solution could be a two unit structure with a dedicated scalar unit just like in vector processors.

References

[Abolhassan 93] Abolhassan, F., Drefenstedt, R., Keller, J., Paul, W., Scheerer, D.: “On the Physical Design of PRAMs”; Computer Journal, 36, 8 (1993), 756-762.

[Agarwal 93] Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B., Yeung, D., D'Souza, G., Parkin, M.: “Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors”, IEEE Micro, June 1993, 48-61.

[Alverson 90] Alverson, R., Callahan, D., Cummings, D., Kolblenz, B., Porterfield, A., Smith, B.: “The Tera Computer System”; Proceedings of the International Conference on Supercomputing, Amsterdam, The Netherlands (1990), 1-6.

[Brent 74] Brent, R. P.: “The parallel evaluation of general arithmetic expressions”; Journal of the ACM, 21, 201-206.

[Burnett 70] Burnett, G, J., Coffman, E.G.: “A Study of Interleaved Memory Systems”; AFIPS Conference Proceedings SJCC, 36 (1970), 467-474.

[Chang 91] Chang, P., Mahlke, S., Chen, W., Warter, N., Hwu, W.: “IMPACT: An architectural framework for multiple-instruction-issue processors”; Proceedings of the 18th Annual International Conference on Computer Architecture, Toronto, Canada (1991), 266-275.

[Enslow (74)] Enslow, P. H.: “Multiprocessors and parallel processing”; John Wiley&Sons / New York (1974).

[Feldman 92] Feldman, Y. , Shapiro, E.: “Spatial Machines: A More Realistic Approach to Parallel Computation”; Communications of the ACM, 35, 10 (1992), 61-73.

[Fisher 81] Fisher, J.: “Trace Scheduling: A technique for global microcode compaction”; IEEE Transactions on Computers, C-30, (1981), 478-490.

[Fisher 83] Fisher, J.: “Very long instruction word architectures and ELI-512”; Proceedings of the 10th Annual International Symposium on Computer Architecture, USA (1983), 140-150.

[Forsell 94] Forsell, M.: “Are multiport memories physically feasible?”, Computer Architecture News, 22, 4 (1994), 47-54.

[Forsell 96a] Forsell, M., Leppänen, V., Penttonen, M.: “Efficient Two-Level Mesh based Simulation of PRAMs”; Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks, Beijing, China (1996), 29-35.

[Forsell 96b] Forsell, M.: “Minimal Pipeline Architecture—an Alternative to Superscalar Architecture”; Microprocessors and Microsystems, 20, 5 (1996), 277-284.

[Forsell 97] Forsell, M.: “MTACSim—A Simulator for MTAC”; In preparation, Department of Computer Science, University of Joensuu, Finland (1997).

[Fortune 78] Fortune, S., Wyllie, J.: “Parallelism in Random Access Machines”; Proceedings of 10th ACM STOC (1978), 114-118.

[Gajski 83] Gajski, D., Kuck, D., Lawrie, D., Sameh, A.: “CEDAR-A Large Scale Multiprocessor”; Proceedings of International Conference on Parallel Processing (1983), 524-529.

[Gottlieb 83] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M.: “The NYU Ultracomputer - Designing a MIMD, shared-memory parallel machine”; IEEE Transactions on Computers, C-32, (1983) 175-189.

[Hennessy (90)] Hennessy, J., Patterson, D.: “Computer Architecture: A Quantitative Approach”; Morgan Kaufmann Publishers Inc. (1990).

[Hillis (85)] Hillis, W. D.: “The Connection Machine”, The MIT Press, (1985).

[Janik 97] Janik, K., Lu, S., Miller, M.: “Advances of the Counterflow Pipeline Microarchitecture”; to appear in Proceedings of the Third International Symposium on High-Performance Computer Architecture, (1997).

[Johnson 89] Johnson, W.: “Super-scalar processor design”; Technical Report No. CSL-TR-89-383, Stanford University, USA (1989).

[Jouppi 89] Jouppi, N. P., Wall, D. W.: “Available Instruction Level Parallelism for Superscalar and Superpipelined Machines”; Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Operating Systems, Boston, USA (1989), 272-282.

[Karp 69] Karp, R. M., Miller, R. E.: “Parallel Program Schemata”; Journal of Computer and System Sciences, 3, 2 (1969), 147-195.

[Keller 94] Keller, J., Paul, W., Scheerer, D.: “Realization of PRAMs: Processor Design”; Proc. of WDAG '94, 8th International Workshop on distributed Algorithms, Terschelling, The Netherlands (1994), 17-27.

[Leighton (91)] Leighton, T. F.: “Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes”; Morgan Kaufmann, San Mateo (1992).

[McColl (92)] McColl, W. F.: “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), 333-387.

[Moore (96)] Moore, S.: “Multithreaded Processor Design”; Kluwer Academic Publishers, Boston (1996).

[Nicolau 84] Nicolau, A., Fisher, J.: “Measuring the parallelism available for very long instruction word architectures”; IEEE Transactions on Computers, C-33, (1984), 968-976.

[Patterson 82] Patterson, D., Sequin, C.: “A VLSI RISC”; Computer, 15, 9 (1982), 8-21.

[Pfister 85] Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., Weiss, J.: “The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture”; Proceedings of International Conference on Parallel Processing (1985), 764-771.

[Prechelt 93] Prechelt, L.: “Measurements of MasPar MP-1216A Communication Operations”; Technical Report 01/93, Universität Karlsruhe, Karlsruhe, Germany (1993).

[Ranade 87] Ranade, A. G., Bhatt, S. N., Johnson, S. L.: “The Fluent Abstract Machine”, Technical Report Series BA87-3, Thinking Machines Corporation (1987).

[Schwarz 66] Schwarz, J. T.: “Large Parallel Computers”; Journal of the ACM, 13, 1 (1966), 25-32.

[Schwarz 80] Schwarz, J. T.: “Ultracomputers”; ACM Transactions on Programming Languages and Systems, 2, 4 (1980), 484-521.

[Sproull 94] Sproull, R. F., Sutherland, I. E., Molnar, C. E.: “Counterflow Pipeline Processor Architecture”; IEEE Design and Test of Computers, 11, 3 (1994), 48-59.