Realization of PRAMs: Processor Design*

Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer

Universität des Saarlandes, Computer Science Department
Im Stadtwald, 66123 Saarbrücken, Germany

Abstract. We present a processor architecture for SB-PRAM, a parallel machine with shared address space and uniform memory access time. The processor uses a reduced instruction set and provides in hardware mechanisms for the emulation of shared memory: random hashing to avoid hot spots, multiple contexts with regular scheduling to hide network latency and fast context switch to minimize overhead. Furthermore it provides hardware support for parallel operating systems and for the efficient compilation of parallel high level languages. We give technical data for a prototype VLSI implementation with a floating point unit.

1 Introduction

Parallel programming imposes more burdens on the programmer than programming in a sequential setting, but is necessary as long as automatic parallelization does not show satisfying results for all problem fields. Programming with the view of a shared memory has become popular, because it frees the programmer at least from the mapping of data and from programming communication between processors (or processes).

For small scale machines, a physical shared memory is possible and several machines of this kind have been built, e.g. Encore Multimax and Alliant FX/8 [2]. For larger numbers of processors however, a physical shared memory quickly becomes a bottleneck, and hence these architectures follow another idea. Memory is physically distributed into modules, the address space is mapped onto modules by a hash function h. If a processor wants to read the content of address x it sends a request to module $h(x)$ to send back the content of the cell to which x is mapped. If all processors access memory at the same time, the hash function should be such that with very high probability hot spots are avoided. A hot spot is the event that many requests are sent to one memory module, which leads to large access times.

Even if no hot spots occur, network latency makes access to remote modules slow. Architectures like DASH or KSR1 [2] try to avoid access to remote modules by employing caches and allowing multiple copies of cells. This however causes a variety of access times, furthermore cache coherency must be ensured. We will not use caches. If caches are used in our design, they should be at the memory modules and cache lines should be one word long [3].

* This work was supported by the German Science Foundation (DFG) in SFB 124.

The latency can be hidden by running several jobs or *virtual processors* on each physical processor. A job that accesses memory is de-scheduled until the answer from the remote module is back. This requires a minimum number of ready-to-run processes. Valiant calls this *parallel slackness* [8], Smith used this idea for the HEP and the TERA [3], Abolhassan et. al. [1] used this idea in an architecture called SB-PRAM when investigating whether PRAM emulations are feasible with todays technology. There the access time is uniformly large which makes scheduling much simpler.

We will present a processor architecture for the SB-PRAM. There were several design goals:

– mechanisms necessary for shared memory emulation should be provided in hardware,
– parallel operating systems and compilation of parallel high level languages should be supported,
– a floating-point unit should be supported to be able to run numerically intensive programs efficiently,
– a prototype should be produced in semi-custom technology.

The remainder of the article is organized as follows. In section 2 we will review some features of the SB-PRAM architecture. In section 3 we present the processor architecture with a focus on hardware support for shared memory emulation and for parallel operating systems (process management, memory protection). Section 4 presents the concept of logical processors which supports the efficient execution of programs in a process oriented programming environment. In section 5 we sketch the processor board.

2 SB-PRAM Architecture

The SB-PRAM is an example of the shared memory emulations described in the introduction. Here we will sketch some of its features necessary to understand the processor architecture. A more detailed description can be found in [1].

The SB-PRAM uses a linear hash function of the form $H(x) = a \cdot x \bmod m$ where m, the size of the shared address space, is assumed to be a power of two. The factor a is an odd integer between 1 and $m-1$, that is chosen randomly before the start of an application. The module $h(x)$ that contains address x is determined by $h(x) = \lfloor H(x)/(m/p) \rfloor$, if p is the number of memory modules. Hence, the module number is stored in the $\log p$ most significant bits of the binary representation of $H(x)$. The lowermost bits give the local address within one module.

In theoretical investigations, polynomials of degree $O(\log p)$ over $\mathbf{Z}/P\mathbf{Z}$, with P a prime larger than m, were taken as hash functions. Our hashing scheme has the advantage that H is bijective (no secondary hashing at memory modules) and can be evaluated fast. The performance in practice seems to be equivalent or better [4].

The SB-PRAM supports an operation called *multiprefix*, which is an extension of parallel prefix [5]. We assume that processors are ordered. Assume a shared memory cell x with content d_0 and processors $i_1, \ldots, i_k$ ($i_j < i_{j+1}$) that execute multiprefix with an associative operator $\circ$ and data $d_1, \ldots, d_k$. Then at the end of the operation, the content of cell x will be $d_0 \circ d_1 \circ \cdots \circ d_k$, and processor i_j will receive a value $d_0 \circ d_1 \circ \cdots \circ d_{j-1}$. The SB-PRAM network supports operators ADD, MAX, AND, OR. There is also a restricted version called *sync* where only cell x is changed, but no values are returned.

All computations necessary to implement multiprefix take place in the network, the execution time of the operation is not longer than for a load instruction. Several prefix operations of disjunct processor groups on different cells are allowed simultaneously, hence the name multiprefix.

3 Processor Architecture

As described above we use the concept of *virtual processors (vP)* to hide network latency. There are two concepts of scheduling the processes. In the first concept one process remains active until it wants to access the shared memory, or its time slice has ended. Then this process is de-scheduled. In case of a shared memory access it is appended to the waiting–queue. In case of a timeout it is appended to the ready–to–run queue. In either case a process from the ready–to–run queue is activated. When the answer to the request returns from shared memory, the corresponding process is moved from the waiting–queue to the ready–to–run queue. FIFO queues are sufficient because the routing algorithm guarantees that the order of requests is not changed by the network. One major concern of the SB–PRAM design was synchronous execution at assembler instruction level. Hence we cannot use the above concept and use instead a strategy where a fixed number C of processes are scheduled in a *round–robin manner*. After execution of one instruction, control is always switched to the next process, independently of the occurrence of memory access; i.e. $vP_{i+1 \bmod C}$ follows vP_i, where $0 \leq i \leq C-1$. Control for the latter mechanism is much simpler than for the first one. However it requires a context switch after each instruction and demands that all instructions have identical execution time.

The instructions are executed in an instruction pipeline. At the same time step there are several $vP's$ in different stages of the pipeline. As a consequence of the demand for identical execution time it is not possible to skip pipeline stages even if they are not necessary. Another effect of this demand is that all data, addresses and busses have the same format of 32 bit. It follows that we need only one opcode format and that only single–precision floating–point instructions are supported.

We give every *physical processor (pP)* its own local program memory, i.e. we use a *Harvard–architecture*. Holding programs in the global shared memory would at least double network traffic and would either increase C because of access time or require an opcode prefetch. One program memory per pP implies that all $vP's$ work on an identical program. However it is possible to branch depending on the vP–number.

One vP is specified by the content of its register file (including special registers like program–counter PC and status–register SR). The ALU (arithmetic logic unit) and the decode/control unit can be shared among the $vP's$ of one pP.
The number of necessary $vP's$ to hide the network latency is determined by the number of network stages and by the routing algorithm. Theoretical investigations [7] and simulations [4] have shown that $C = c \cdot n$ $vP's$ (with 2^n the number of physical processors) are sufficient to hide network latency. Using *delayed load* allows to halve c from $c = 6$ to $c = 3$. *Delayed load* means that the value read out of a memory cell is not available in the next instruction but in the next but one. In our planned prototype we have $2^7 = 128$ physical processors. This leads to $C = 21$ $vP's$ per pP. We only allow a pP to run alternatively 8, 16, 24 or 32 $vP's$ to simplify the necessary counting mechanism.

3.1 Instruction set

Because of the simplicity and because of the restricted opcode length we have implemented a *reduced instruction–set*. As a basis we have chosen the instruction–set of the Berkeley–RISC I [6]. However, we provide more arithmetic. In addition to integer addition and subtraction, bit operations and shifts and rotates (both logical and arithmetical), we support *rightmost* (integer part of the logarithm to base 2 of an integer argument) and multiplication of integers. The user can choose between either the 32 most significant or the 32 least significant bits of the product as the result.

Branch instructions can be relative or absolute, either unconditional or depending on floating–point or integer conditions. Push and pop instructions allow pre–addition of a constant and optional post–increment.

We use 3–address–instructions (two operands, one result). Operands are either a register or a 13–bit immediate constant. Computation with memory–cells as operand or result is impossible. This is called *load–/store–architecture*.

3.2 Register–files

We have 32 registers per vP, each 32 bits wide. Register R_0 means either the value 0 or the content of the status–register SR, depending on the executed instruction. Register R_1 is the program–counter PC. Registers $R_2 \ldots R_{31}$ are universal registers, i.e. they can hold an integer, a floating–point number, a memory address or can be used as stack–pointer.

Because a *full–custom design* would have been too expensive, we restricted ourselves to a *semi–custom design*: a *sea-of-gates* architecture by MOTOROLA in a $0.7\mu m$ technology. We chose a master with about 80,000 gate equivalents and about 230 signal pins.

To implement the 32 registers with 32 bits each for 32 $vP's$, $32^3 = 32,678$ 1–bit register cells are needed. Using flip–flops to realize these cells requires

$8 \times 32,768 = 262,144$ gate–equivalents. Implementing the register cells by on–chip RAM modules would still need approximately 100,000 gate–equivalents. Therefore the register files are realized by fast off–chip static RAM's.

In the chosen technology it is not possible to implement bidirectional busses with the desired speed. Because of pin–limitation it is also not possible to replace each bidirectional bus by two unidirectional busses. Hence, the busses have to be multiplexed. Each processor cycle is therefore divided into two half–cycles.

During the execution of one instruction one needs up to four accesses to the register RAM's, two for loading the operands, one for storing the result, and one for a possibly returning answer to a load instruction from the network. Because of timing limitations only one access per half–cycle is possible. Therefore the register RAM's have to be either dual–ported or one has to use two single–port RAM's and partition the registers of the $vP's$ among them. We choose the second option (two RAM–chips $16K \times 32$, 20 ns access time) because there are no appropriate dual–ported memories available. Partitioning the register files into two groups is easy as $vP's$ with odd and even numbers alternate.

3.3 Memory access

As already mentioned we use a linear hash function which needs an integer multiplication with the hash factor a. During the initialization routine the hash factor is loaded into a special register on–chip.

Although we use *delayed load*, there is a slight chance that there is too much traffic in the network. If the answer to a load request is not returned in time the processor is frozen.

The internal memory of the network interface to store returning data is limited. To avoid additional waiting situations because of a full buffer in the network interface, data coming from the network should be passed on as soon as possible. However the processor has the necessary control information (e.g. the destination register number) for returning load packets only available when the corresponding vP is in the appropriate pipeline stage. Therefore an off–chip FIFO queue buffers the data coming from the network interface. The necessary length of this FIFO queue is limited to twice the number of $vP's$ because the processor is frozen if the answer to a request has not returned after two processor rounds, and no more packets can be generated within this time.

Instructions that access shared memory are load, store, push, pop, multiprefix, sync and jump–to–subroutine (push PC onto stack, post–increment stack–pointer). The push and pop instructions allow to add a 13 bit integer immediate value before the multiplication with the hash factor and optionally a post–increment. This allows fast copying of memory sections which may overlap. For load, store, multiprefix and sync the address is computed by adding the content of a register and a 13 bit immediate constant.

We provide a mechanism that simplifies access to private data. Each vP of a pP works on the same program. The processes need private variables which are also held in the shared memory. For performance reasons it should be avoided that a vP must use an additional compute instruction to add the base–address each time it accesses these variables. This base–address is held by each vP on–chip in the base–register ($BASE$). An access where adding $BASE$ is desired is specified by setting the most significant address bit *msb*, i.e. bit 31.

3.4 Protection

We provide two mechanisms to protect system resources, e.g. disks, from uncontrolled access by distinguishing between *user*– and *supervisor*–mode. First, in user–mode a vP can only access its own universal registers. To access any other register, e.g. $BASE$ or SR, or to execute an instruction that supports context switch (see section 4) the vP has to be in supervisor–mode.

The other mechanism aims at protection of different memory areas against each other. This is an essential demand for multitasking as one user–program has to be protected against accesses from other user–programs.

The global address space is partitioned into areas. The size of each area is a multiple of 64KWord. Hence, start and end of each area can be specified by two 16–bit addresses. Every vP has a register $PROT$ on–chip which contains these addresses. During the execution of each instruction that accesses memory these bounds are checked while the vP is in user–mode. If the check fails the execution is invalidated. In multitasking it is possible that data of one program are held in different memory areas each time the program is started. It is possible to configure the pP such that the start address, held in $PROT$, is added to the computed address. This saves one compute instruction for each access.

Trying to execute a prohibited instruction or to access a protected memory area results in a branch to the illegal–exception routine as described in subsection 3.6.

3.5 Floating–point

For solving numerical problems efficiently a floating–point unit is mandatory. Because of the uniform data format of 32 bits only instructions with single precision are supported. Commercial floating–point co–processors at suitable speed are available. However, more complex floating–point operations as $\sin(x)$, e^x, $\sqrt{x}$ cannot be used even if the co–processor supports them, because of the fixed number of pipeline stages.

The timing for writing results from chip into the off–chip register RAM is critical. Therefore introducing some multiplexing–scheme to incorporate the co–processor would slow down operating frequency. Another 32 bit–wide data–bus would be necessary. Other problems with commercial co–processors are additional pins required to configure the co–processor and to bring floating–point conditions onto the processor chip. The second case could lead to a delayed branch instruction.

The above considerations give reason for the decision to implement a floating–point unit on–chip. We use IEEE standard 754 for single precision floating–point numbers. We provide addition, subtraction and multiplication with two floating–point operands. These operations are also available with taking the absolute value of the result. Two conversion instructions allow fast transition from and to integer computations.

We use the flags proposed in the IEEE standard and allow the appropriate conditional branch instructions. Several flags allow to enable or disable the initiation of interrupt routines, e.g. in case of overflow, underflow, infinity, invalid operations.

The largest part of the floating–point unit is the multiplier for the mantissa. To save chip area, we implement this multiplier with the help of the already existing integer multiplier, which additionally is used for evaluation of the hash function.

3.6 Exceptions

Under the term *exception* we collect all actions disturbing the program flow, e.g. a floating–point trap caused by an overflow or a disk interrupt. We have sixteen exception levels with hardware–reset as level 0. All but level 0 are maskable. We distinguish three classes:

common external exceptions: These seven exceptions are caused by some event off–chip, e.g. disk, timer, bus–error, interrupt from another pP.
Common means that all $vP's$ have to execute the exception routine. Execution is always started with vP_0 regardless of the time when the appropriate signals were activated off–chip.

common internal exceptions: These three exceptions are caused by an internal event.
We have a counter on–chip, counting complete instruction rounds. If it reaches the value 0 all $vP's$ branch to the counter routine.
There is an instruction that allows one vP to force all other $vP's$ of this pP to the corresponding exception routine.

private internal exceptions: These exceptions are caused by the instruction the vP tries to execute.
An *illegal exception* occurs if the vP tries to execute a prohibited instruction, or tries to execute an opcode which does not code an instruction, or tries to access a protected memory area.
The *data exception* is activated when the loaded data has an error, i.e. a parity error or some problems in the network.
The *floating–point exception* is executed if the operation results in an overflow, underflow, infinity or the computation was not possible because one of the operands was not a legal number. One can control the initiation of the exception more exactly by some mask flags in the SR.
The *context exception* is activated if an instruction on contexts (see section 4) fails, e.g. the instruction tries to create a new context if there is no one available.
The *system–call* allows the user program to execute operating system routines where supervisor mode is necessary (e.g. disk access).

External exceptions have higher priority than internal ones and common internal exceptions have higher priority than private internal exceptions. An enabled common exception is always taken, even the $vP's$ which are in a private internal exception routine branch to that routine. Upon entering an exception routine the supervisor bit for this vP is set. It is also possible to execute the exceptions recursively.

4 Logical Processors

Using the concept of virtual processors, a user of our prototype with 128 pP's will see a total number of $128 \times 32 = 4096$ processes. Some applications however require dynamic creation of large numbers of processes at runtime. As swapping the register files without hardware support is too time consuming, we provide an efficient mechanism to handle large numbers of processes. The obvious solution is creating new processes by a special instruction which adds a new vP. However, this would contradict our principle of a simple instruction pipeline of fixed depth and should therefore be avoided.

Instead, we introduce another hierarchical level, the *logical processors, lP*. The number of used $lP's$ can dynamically be changed at execution time from 1 to 32 per vP. The switch from one lP to the next one is not done automatically, as is the case for the $vP's$, but must be programmed explicitly.

As the register files are held off–chip, it is easy to implement 1024 contexts instead of 32 per pP. A context switch now consists mainly of exchanging PC and SR of the actual process, which are held on–chip, with PC and SR of the new process, which are swapped to the off–chip register RAM. An exchange of the universal registers is not necessary, as these are always held in the register RAM. Only the base pointer that addresses a specific context within the register RAM has to be changed.

If context switch is to be completed within one instruction, we run into the problem that we are allowed to load two operands from the register RAM (PC and SR of the new context), but to store only one result of the actual context. Storing the PC is mandatory and therefore chosen. If the SR of the actual context contains essential information, it must be stored into the register RAM with an additional instruction before the explicit context switch.

There are two mechanisms to update the base pointer to the register RAM, which we will describe in the following subsections.

4.1 Context switch with hardware support

As already mentioned every vP can manage up to 32 $lP's$. There are three possible states for every lP. There is exactly one *active* lP. A *sleeping* context has at least initialized PC and SR in the register RAM and is waiting to become active. The third state is *dead*, i.e. the contents of its register file are undefined. It is not possible to switch off the active context if there is no other *sleeping* context, because then this vP would no longer exist. The following instructions allow to change the state of $lP's$.

CREATE: A *dead* context is chosen and its PC, SR and, if necessary, further registers in its register file are initialized. The state of that context is changed from dead to sleeping. If there is no dead context available processor control is switched to an error–routine.

HOP: Switch from the active context to the next sleeping if there is one, otherwise branch to the error–routine. The former active context is appended to the queue of sleeping contexts.

DIE: Analogously to `HOP`, the next sleeping context is chosen and gets active. However, the former active context is appended to the queue of dead contexts.

Administration of active, sleeping and dead contexts is handled by a 32–bit vector per vP. A one in the vector represents a sleeping vP, a zero a dead vP. The vector is always rotated at HOP and DIE instructions in such a way that the new active context occupies position 0. The number of the active lP is held in a 5–bit register. The hardware for the context control unit has to perform several tasks, e.g. find the first 0 in the vector (`CREATE`), find the first 1 (`HOP`, `DIE`), rotate the vector by the appropriate distance, update the number of the non–dead $lP's$. We omit further details because of space limitations.

4.2 Explicit context switch

The mechanism given by the instructions `CREATE`, `HOP`, and `DIE` provides easy access to a large number of contexts with a scheduling strategy that should suffice for many cases. For applications however where this mechanism is not flexible enough, we provide the possibility to explicitly manage creation, switch and termination of contexts. The necessary queues must be maintained in shared memory and managed by the user. The number of the active context is still held in the 5–bit register, as in the hardware supported mechanism. When a context switch occurs, the number of the new context must be specified. This is done in a 5–bit field within the SR. The desired number cannot be specified in the opcode because it will not be known at compile time if one uses dynamic scheduling. The following instructions are used.

PUTSEC, GETSEC: These instructions allow to access the register file of the lP specified in the SR.

HOPSEC: The active context is deactivated, control is switched to the context specified in the SR. The PC and SR are loaded from the register file into the processor chip. The PC of the deactivated context is stored into the corresponding cell of the register RAM.

5 Processor Board

5.1 Functional blocks

We use one board per pP. On this board the following devices are located.

- The network interface and the FIFO queue to buffer returning data connect the processor board to the network.
- The processor boards are connected to the host by a bus. Via this bus the host can load the program memory. It can also be used to access the disks, which can be necessary if the processor board fails but the disk itself remains accessible.
- A serial interface allows to connect a terminal to the processor board. This simplifies debugging of single boards. This device also contains a timer unit which can be used to interrupt the processor.
- There is also a small local memory. It is helpful during the hardware debugging phase. Normally it is used for accessing the hard disks (see subsection 5.2).

These devices are accessed via a local address/data bus. Since the addresses for the shared memory are hashed we need a second unhashed address space for the memory map of the devices. Therefore for each instruction that accesses memory (load, store, push, pop, jump–to–subroutine) we have two versions, e.g. `POPG` and `POPL`. The instructions with postfix `G` access the global shared memory, `L` access the local on–board address space.

These local load instructions are not *delayed*, i.e. the vP can use the result in the following instruction. In analogy to global loads the result of the local load is buffered in a FIFO queue until the requesting vP is in the appropriate pipeline stage. A local load following a global load is prohibited and leads to an exception, because the two results would have to be stored in the same time slice.

The devices on the board should only be accessed within operating system routines. With a protection mechanism similar to the one from subsection 3.4 it is possible to prohibit executing local instructions in user–mode.

5.2 Disks

Disks and the corresponding controller ports are 8 bit wide. Accessing these ports using processor busses would need ten processor instructions per transmission of a 32 bit word (for reading/writing and the appropriate shifting and merging). We use the 32 bit wide local memory as intermediate memory between the shared memory and the disks. For an efficient handling we need DMA (direct memory access) transmission between disk controller and local memory. We use an FPGA (field programmable grid array) to realize the control logic for the board. The necessary logic for the DMA is also implemented on this FPGA. We have an address register, an address incrementer, a length register, a register to control the disk operation, a register reflecting the status of disk operation, the shifting and multiplexing scheme for data.

We reach a transmission rate between disk controller and local memory of one 32 bit word every fourth processor cycle per disk. This exceeds the transmission rate of the disks used. Data transmission between local memory and shared memory is managed by the processor. Using loop unrolling two instructions suffice to transfer a 32 bit word: `POP` with pre–decrement from shared memory and `PUSH` with post–increment to the local memory. Unrolling is easy because fixed size blocks are transferred. The number of $vP's$ which are involved in the data transfer determines the bandwidth.

Storing data to disks can be caused either by an explicit statement in a program or by a statement of an interrupt routine (e.g. timer interrupt to initiate writing a checkpoint to disk). If the desired bandwidth cannot be reached by the *executing vP*, it interrupts the other $vP's$ of this pP. The interrupted $vP's$ branch depending on their process number to the disk routine and work as *assistant* $vP's$. These $vP's$ transfer the data from shared memory to the local memory and return to the previously done work. The *executing vP* initializes the DMA address register and the length register on the FPGA. Writing to the DMA control register starts transfers from local memory to the disk controller. The *executing vP* completes the transmission routine.

Data transfer from disk to shared memory is done in a similar way. After initiating the transfer by writing to the DMA registers, the *executing vP* polls the DMA status register until it recognizes that the desired data have been transferred to local memory. Then the transfer is completed by transporting the data to the shared memory, if necessary with the help of some *assistant vP's*.

Because disk bandwidth is critical for applications like databases we use two disks per processor board. These disks are mechanically mounted on the processor board. This is possible because todays available disks are small and power consumption is low. These disks have a capacity of at least 200 MByte and reach a data transfer rate of 5 MByte/s.

References

1. F. Abolhassan, J. Keller, and W. J. Paul. On the cost–effectiveness of PRAMs. In *Proc. 3rd Symp. on Parallel and Distributed Processing*, pages 2–9. IEEE, Dec. 1991.
2. G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 2nd edition, 1994.
3. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. 1990 Internat. Conf. on Supercomputing*, pages 1–6. ACM, 1990.
4. C. Engelmann and J. Keller. Simulation-based comparison of hash functions for emulated shared memory. In *Proc. PARLE '93, Parallel Architectures and Languages Europe*, Lecture Notes in Comput. Sci. No. 694, pages 1–11. Springer, June 1993.
5. R. E. Ladner and M. J. Fisher. Parallel prefix computation. *J. Assoc. Comput. Mach.*, 27(4):831–838, Oct. 1980.
6. D. A. Patterson and C. H. Sequin. A VLSI RISC. *Comput.*, 15(9):8–21, 1982.
7. A. G. Ranade. How to emulate shared memory. *J. Comput. System Sci.*, 42(3):307–326, 1991.
8. L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.

This article was processed using the LaTeX macro package with LLNCS style