Building the 4 Processor SB-PRAM Prototype[*]

Peter Bach, Michael Braun, Arno Formella, Jörg Friedrich, Thomas Grün, Cédric Lichtenau
Universität des Saarlandes, FB14 Informatik, 66041 Saarbrücken, Germany
E-mail: {pb,mbraun,formella,jf,gruen,cls}@cs.uni-sb.de

Abstract

The SB-PRAM *is a massively parallel, uniform memory access (UMA) shared memory computer. The main ideas of the design are multithreading on instruction level, hashing of the address space, and combining in the butterfly network. We have built a first research prototype with 4 physical processors, thus 128 virtual processors, to demonstrate the feasibility of the concept.*

The programming environment consists of a FORK compiler for specifying PRAM programs, an extended C compiler, and the P4 library. The machine runs a parallel operating system which provides program execution and I/O system calls. The SB-PRAM *allows for efficient programs with predictable performance. Some examples are presented. The 4 processor prototype is the first step towards a 128 processor machine for which we are adapting the existing hardware.*

1. Introduction

The PRAM is a well-established, parallel machine model widely used in theoretical computer science [16]. The main characteristic of the model is that the parallel operating processors have concurrent access to the entire memory with unit access time. There has been much research on how to implement this machine model in real hardware (see [15, 32] for a survey). One of the most promising approaches is the fluent machine [27, 28] by Ranade. This approach has been revised and led to the SB-PRAM design [1], a uniform memory access (UMA) shared memory multiprocessor which conforms to the priority concurrent read concurrent write PRAM model (priority CRCW PRAM).

The SB-PRAM consists of n processor modules and an equal number of memory modules. The processors are connected by a butterfly network to the memory modules. A memory reference of a CPU is turned into a network packet that travels through the network to the appropriate memory module. In case of a load request a response packet is sent back from the memory module to the processor.

[*]This work was supported by the German Science Foundation (DFG) under contract SFB 124, TP D4.

To get high performance, one must pay attention to some crucial design topics. Above all, the network latency needs careful treatment. The latency is hidden by two means: first, a physical processor simulates several *virtual processors* or threads; second, a delayed load instruction is used, i.e., the destination register of a load is updated after the *next* instruction. Another issue is to hash memory addresses produced by the processor among the memory modules, which equalizes the load of the modules. Furthermore, packets with the same destination address are combined in the network stages into one packet to avoid memory hot spots. Combining is extended to multi prefix operations on integers.

The concepts underlying the SB-PRAM have not been developed independently from others. The concept of virtual processors in hardware was already used in the Denelcor HEP [31], and is used again in the Tera MTA [2]. The concepts of hashing and combining were already used in the NYU Ultracomputer [12] and IBM RP3 [25]. NYU Ultracomputer, IBM RP3, Tera MTA, and Stanford DASH [24] (a multiprocessor with cache-coherent virtual shared memory) also have hardware support for parallel prefix operations, although DASH is restricted to increment/decrement. The aim of the SB-PRAM is to bring all these concepts together in a single machine.

Besides algorithms that have chiefly been developed for the theoretical PRAM machine model, many applications of commercial interest will perform well on a PRAM type computer. Among them are data base management systems [11], telecommunications applications like ATM switches [8], Video-on-Demand servers [10], and ray tracing applications [3].

In the next two sections, we report on the running 4 processor prototype (4-SB-PRAM). First, we describe the hardware implementation and the physical organization of the machine. Then, we discuss the system software and the programming environment of the machine and give a brief overview of the application programs that have been run on the prototype. Finally, we give an outlook to the planned 128-SB-PRAM and sketch the modifications in the implementation of this machine.

2. The 4-SB-PRAM hardware

The overall structure of the SB-PRAM is depicted in Figure 1. The core of the machine is the network which connects the processor modules and the memory modules. The machine is controlled by a host which is connected through a host interface. In the following we will describe these parts in more detail.

2.1. Processor

The processor was designed [19] in a sea-of-gates ASIC technology. Its Berkeley RISC like instruction set supports integer arithmetic and logic operations on 32-bit operands. Additionally, single precision floating point addition and multiplication—compliant to the IEEE standard—are implemented in hardware. More complicated operations and double precision have been omitted due to chip space limitations in the technology.

Load and store instructions can be of two types: *local* or *global*. The local instructions access memory and peripherals on the processor board while the global ones access shared memory through the network. For accessing global memory, the processor also supports single cycle parallel prefix instructions. Several groups of processors may perform prefix operations on distinct memory locations in parallel, hence the name "multi prefix".

When accessing shared memory, logical addresses are translated to physical addresses in two steps. First, if the most significant bit of the original logical address is set, the contents of a special register `hi_base` is added; otherwise `lo_base` is added. This provides a simple means to map private and shared data to different address spaces [13]. The second step realizes a linear hash function which has been considered sufficient for hashing [9]. The sum is multiplied by a 32-bit constant and the lower 32 bits of the result form the physical address.

Currently, a physical processor schedules 32 virtual processors in a pipelined manner on instruction level. Each virtual processor works on its own register set consisting of 32 registers. Due to its size the register file is implemented off-chip.

2.2. Processor board

Figure 2 depicts the functional blocks of the processor board. Program memory is accessed by a separate bus (harvard architecture). The local bus of the processor board supports various functions: *i)* It enables the processor to access its local memory, the registers of the control FPGA, on-board I/O devices (two SCSI controllers, a serial line interface) as well as devices attached to the extension port; *ii)* The bus provides the same functionality as in i) to the host. Particularly, some special registers of the FPGA controlling the state of the board may be set; *iii)* The FPGA may perform a DMA transfer on the local bus between local memory and two SCSI hard disks. It achieves a data rate of 32 MByte/s without interfering with the processor. Local memory is currently only visible to the operating system not to the compiler.

Access to shared memory is handled by the sortnode, a special ASIC which realizes the access to the network. Its functionality belongs to the network and will be discussed in Section 2.5.

2.3. Global shared memory

The global shared memory consists of 4 modules equipped with 16 MByte RAM each. Two independent modules are integrated into one board. The memory consists of standard DRAM SIM modules. Memory requests from the network may arrive every 30 ns in the final design. With a DRAM cycle time of 120 ns the memory becomes a bottleneck. The problem is solved by splitting a memory module into four banks of 4 MByte each. The hash function distributes the accesses uniformly over the four banks of a module.

The memory control of a bank is implemented in an Actel FPGA. Besides refresh and address generation for ordinary read/write operations, it also performs read-modify-write operations which are needed by the multi prefix instruction. Another FPGA chip controls the communication between the memory board and the connected network board. It also provides small queues for buffering incoming data from the network. All memory modules may be accessed by the host at any time using a separate data path.

2.4. Host interface

The host computer is a Sun SPARCstation10 with an SBus expansion bus. The host interface consists of two boards, the SBus card and the PRAM interface board. The SBus card simply buffers and transmits data for the second board. This choice was made to keep debugging simple. At the PRAM interface the data path is split into two independent busses, the processor bus and the memory bus: for ordinary I/O operations, the memory bus would be sufficient, but the processor bus was provided for testing and debugging the processor boards in the initial construction phase. It offers access to the program memory and the local bus of the processor boards.

Since the logical address space is hashed onto physical memory, the host has to employ hashing, too. The hashing is done in hardware by a 32 bit multiplier chip. Clearly, the addresses for the program memory are not hashed. The processor and memory backplanes distribute the signals of the host to the individual boards.

All actions of the host interface are initiated by the host since the interface is an SBus slave device. The SB-PRAM

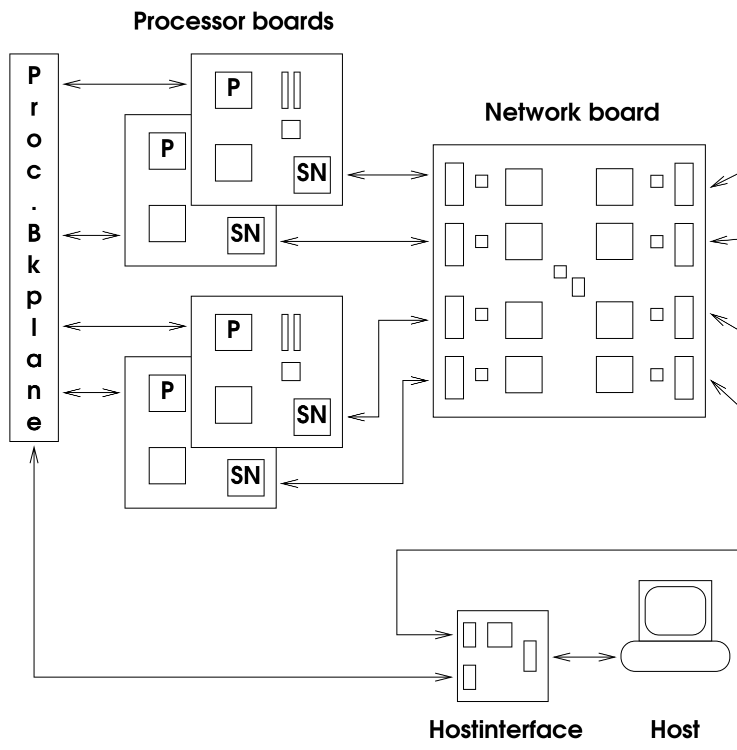

Figure 1: Overall structure of the SB-PRAM

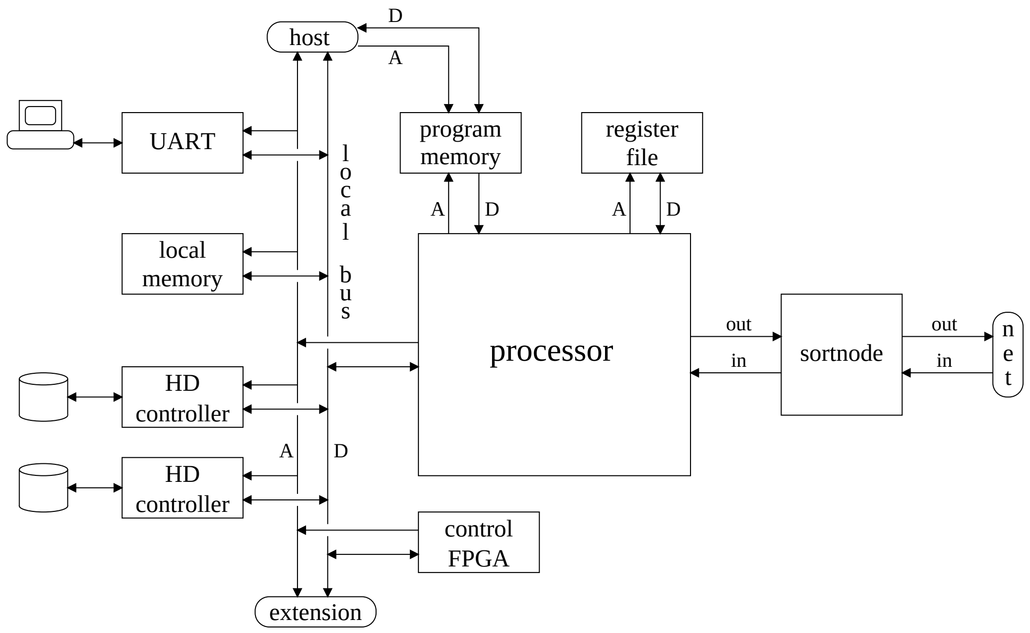

Figure 2: Data-paths of the processor board

is able to interrupt the host on seven interrupt levels. Interrupts can be initiated by the processor boards during normal operation (e.g., I/O handshake) or by the memory boards in case of a memory fault (e.g., parity error) or a network failure. The host interface is accessed via a device driver, which is responsible for mapping SB-PRAM memory to Sun memory and for interrupt handling.

2.5. Network

The connection between the processors and the memory modules of the global shared memory is realized by a butterfly network. Processors may send load, store or multi prefix requests to the memory. They are routed by the butterfly network to their destination memory modules. If the request was a load or a multi prefix operation, an answer is sent back to the requesting processor. In the following, we discuss major aspects of the functionality of the network and detail its implementation for the 4-SB-PRAM.

Each processor is connected to the sortnode ASIC which in turn is connected to a link of the network (cf. SN in Figure 1). The sortnode collects the requests of eight virtual processors and sends the requests sorted by their hashed addresses to the network. After having sent the requests, an additional End-Of-Round packet (EOR) is sent into the network. EOR-packets are sent always, even if there are no requests in a round.

In the SB-PRAM, $n = 2^k$ processors are connected to n memory modules through a $k + 1$-stage butterfly network (cf. Figure 3). A unique path exists from each processor to each memory module. In forward direction, the routing path is determined by the address of the request. If an answer packet is expected, routing information is stored in a FIFO queue in each node. In backward direction, data are routed according to the contents of the queue.

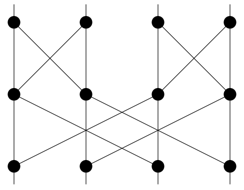

Figure 3: Network topology for $n = 4$

If there are requests accessing the same address within a round, they are combined and only one request is passed to the next stage of the network. This is possible, because the sorted order of the requests, which is produced by the sortnode chip, is preserved in the network [20]. On the backward path, the combined requests are re-duplicated using the information in the FIFO queue.

Due to the combining of requests at most four of them arrive at the memory module, even if all processors access the same address simultaneously. On a store request, the operand of the rightmost path, i.e., of the processor with the highest number, is stored in memory (this realizes the priority PRAM model).

Multi prefix operations as defined in [27] are implemented in the network as a special form of combining. In the forward path two operands i and j are processed according to their multi prefix operation $\odot \in \{and, or, max, add\}$. The result is sent down the network. Furthermore, the left operand i and the operation $\odot$ are stored in a second FIFO queue. On the backward path, the incoming answer s is sent up the left link and $s \odot i$ is sent up the right link. An ALU is needed in the network nodes to perform the computations.

The network operates at a clock frequency four times faster than the processors. As the network needs two cycles to handle a request, a processor utilizing the network at its peak bandwidth can have a speed of at most twice as fast as the network. We assume here that a processor is able to access the global memory via the network in each instruction. However, a utilization of 100% is not possible because conflicts can occur within the network. To keep the protocol between processors, sorting devices, and network nodes simple, we chose cycle times that are multiples of each other.

In each network cycle, packets may be sent along a link. The flow of packets in the network is controlled by valid signals in forward direction and busy signals in backward direction. The inputs of the nodes are buffered by small FIFO queues.

The network nodes are realized in the same technology as the processor. As shown in Figure 4, parts of different nodes are integrated into one chip. This reduces the wiring effort off-chip [7]. Each ASIC represents two complete nodes of the network. The nodes to the processors and to the memory modules need only one link. A first version of the network ASIC was found not to be functional [20], the original network design of the fluent machine [28] was implemented incorrectly. The error was found through simulations, but due to time constraints with the manufacturer, the chip had already been manufactured.

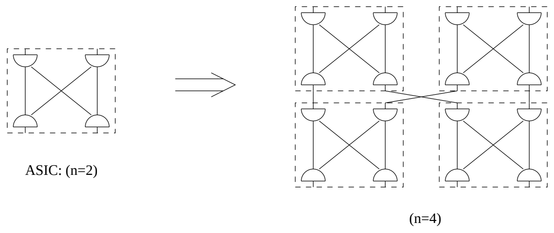

Figure 4: Partitioning of the network.

For the current 4-SB-PRAM prototype, the network is emulated with an FPGA-Design. Due to the complexity, each ASIC is embedded using two FPGAs XC4013, where one FPGA holds the forward and the other FPGA the backward paths and logic. Using FPGAs for the first implementation offered several advantages.

The FPGA XC4013 from Xilinx is a re-programmable and in circuit programmable device. So it was very easy to change the design and the behavior of the network without modifying the hardware. With the implemented FPGA design, we have been able to test the behavior of a network node before manufacturing the ASIC (version 2). In the FPGA, additional logic could be implemented to check correctness of the network's behavior. The 4-SB-PRAM is running with a network clock frequency of 5 MHz which is limited by the FPGA based network board. When moving to ASIC technology, the clock frequency of the system will be increased to 32 MHz.

2.6. Summary

The SB-PRAM project started early in 1990. When it became clear, in 1991, that such a system could be realized, the design of the chips and the boards started. About a dozen persons were involved in the hardware design process. A functional prototype of a 1-SB-PRAM without network was available at the end of 1995. By now, the 4-SB-PRAM is up and running.

3. Software

Software development began when the hardware design started. So, now that the hardware is working, there is already a functional program developing environment and some application programs.

3.1. System software

The system software comprises two compilers, one for the programming language C and one for FORK [21], a C extension especially developed for the specification of PRAM programs. The parallel programming environment of the C compiler [13] is provided by the P4 library [5] and the SBP library. The P4 library consists of functions for writing portable, parallel programs and the SBP library provides functions that are tuned for the SB-PRAM. The assembler and linker are common for both compilers.

Both programming environments use a small set of system calls for interfacing to the operating system PRAMOS. The operating system handles all I/O operations and performs functions that require supervisor permissions, e.g., the creation of processes.

The programs can be executed either on the real 4-SB-PRAM or on a PRAM simulator which has been available early in the project. Since the simulator emulates system calls, it was possible to develop and test application programs while the hardware was not yet available.

The creation of the programming language **FORK** dates back to the year 1990, when it was designed as a language for specifying algorithms for the theoretical PRAM model in an elegant way [14]. Since then, a lot of work has been invested to make it ready for practical use [21, 22]. A functional compiler exists and a library of basic PRAM algorithms is currently under development [23].

The C compiler of the SB-PRAM, called **PGCC**, is an adapted version of the GNU C compiler. Besides the changes to the code generator, there are two important modifications: new storage class identifiers `shared` and `private` have been introduced, and all elementary data types are 32 bit wide. The storage class qualifiers specify whether all processes get their own copy of a variable (`private`) or whether all processes share the same variable (`shared`). Accordingly, the function $shmalloc()$ dynamically allocates space for shared variables whereas the standard $malloc()$ routine reserves private space, all in global memory. Since the processor only supports 32 bit memory accesses, all basic data types are defined 32 bit wide. Hence, all `char` variables need four bytes, but this imposes no severe problems even if I/O is involved.

The *C Standard Library* had to be adapted to the 32 bit data types. The *Mathematical Library* implements basic functions that are not directly available in hardware, like floating point division, and more complex functions, e.g., $sqrt()$ or $sin()$. A *Multi Prefix Library* gives the C programmer direct access to the powerful multi prefix operations. The *Parallel Library* contains a set of efficient parallel data structures and algorithms. Among them are different versions of locks, barriers, and parallel queues. Many of them are based on multi prefix and operate serialization free in constant time. Furthermore, the complete P4 library has been ported.

The assembler generates object files in a slightly extended COFF format. The standard COFF format has an address space of 2^{32} Bytes. This address space had to be adapted to the 2^{32} word address range.

The linker joins the appropriate segments of its input files. The program code is contained in the standard `text` segment, while the usual `data` and `bss` segments are split into a private and a shared part. The non-standard segments `args` and `lddata` provide the command line arguments and further information for the startup code, like the size of the shared heap (`sh_size`), the private heap (`ph_size`), and the private stack (`ps_size`).

The memory map during program execution is shown in Figure 5. It takes advantage of the virtual processor's two address spaces which are distinguished by the uppermost address bit. All shared segments, namely `lddata`, gsdata, gsbss and the shared heap, are arranged in the lower memory half and are mapped to the same location in logical memory for all virtual processors.

For the private segments, namely gpdata, gpbss, args, the private heap, and the private stack, a *private memory frame* (PMF) is set up in the upper memory half. Since the stack and the heap grow in different directions, the starting point of a PMF is located in the middle of the upper address half of a virtual processor. The PMF is mapped onto a different memory range for each virtual processor. The logical address range for a program is contiguous and is limited by the lo_protect and hi_protect registers of the memory management logic which disables accesses outside this range and signals an interrupt.

The operating system PRAMOS shall enable multiple users to work concurrently on the SB-PRAM in the future, but by now, it offers only single user support. The operating system is split up in a host part and an SB-PRAM part. The host is responsible for booting the machine, for user login, and for program execution. After a program is started, the host only acts as an I/O device performing terminal I/O to the user, network I/O via sockets, and file I/O to its own local file system. On the SB-PRAM one “slice” of virtual processors, i.e., one virtual processor per physical processor, is allocated as a “service processor” for PRAMOS.

When loading a user program, both operating system parts work closely together. They communicate over shared memory. The user specifies the program to be executed, the number of processors to be used, and the memory space. Then, the host sends a program load request to the SB-PRAM. The service processors allocate memory and virtual processors, and answer the load request. Then, the host uploads the program and data according to the specified addresses and interrupts the SB-PRAM. In the interrupt service routine, which belongs to PRAMOS, the necessary initializations are performed and the user program starts when the interrupt routine is finished.

Memory is allocated in a contiguous logical block to exploit the memory protection logic of the processor. The number of processors is specified by the number of slices over all physical processors. Thus, the program has always access to all disks of the SB-PRAM.

Besides providing system calls, the operating system must also initiate rehashing [18], if the hash factor was badly chosen. For achieving this, PRAMOS periodically controls how many network packets returned too late.

As long as the SB-PRAM hardware was not available, application programs and even PRAMOS have been developed using a software simulator of the machine. Simulating one virtual processor, it has a performance of about 0.7 MIPS on a Sun SPARCstation20 and is able to emulate system calls.

3.2. Applications

We discuss the following complex application programs and their implementations on the SB-PRAM. From the SPLASH application suites version 1 and 2 [30, 33] we ported LocusRoute, Radiosity, MP3D, and PTHOR. The programs are running on the simulator already and we are currently in the debug phase of the code on the 4-SB-PRAM, so we cannot present exact, proven and comparable performance data of the prototype. Two applications, namely a fish-eye view of graphs and a ray tracer have been developed by ourselves and they are running successfully on the real machine. Furthermore, we investigated data base implementations, Video-on-Demand servers, and an ATM switch as possible applications for the SB-PRAM where the machine promises high efficiency. The description of the programs is very brief and focuses only on some main issues of the implementation.

SPLASH programs. The programs taken from the SPLASH application suite are written in C and use the P4 macro package. After they have been ported *as is* to the SB-PRAM several draw backs of the specific implementation of the benchmarks have been detected. It seems that the benchmark was developed to achieve good performance on machines with physically distributed memory and non-uniform access time. The powerful mechanisms that are provided by the SB-PRAM architecture such as lock free parallel queues, parallel shared memory allocation, and single cycle multi prefix operations allow for a more efficient implementation of the applications as described in the sequel.

LocusRoute is the standard cell global router from the SPLASH-1 application suite. LocusRoute approximates the optimal solution for the global routing problem by enumerating a reasonable subset of all routes between two pins that have to be connected and selects the route that causes minimum costs. The main data structure of LocusRoute is a *cost array* which has a row for each channel and a column for each channel segment. The entries of this array keep track of the number of wires running through each channel segment. Although the original SPLASH implementation provides two levels of parallelism, namely routing of different wires in parallel (wire tasks) and testing different routes between two pins of a given wire in parallel (route tasks), the experiments reported in the benchmark distribution exploit only the wire parallelism by geographic wire partitioning. Because a wire task may generate many route tasks before it can be terminated, the granularity of a route task is usually much finer than the granularity of a wire task. In our implementation, the wire tasks and route tasks are held in central task queues that can be accessed by all processors. Initially, the processors are partitioned into wire processors and route

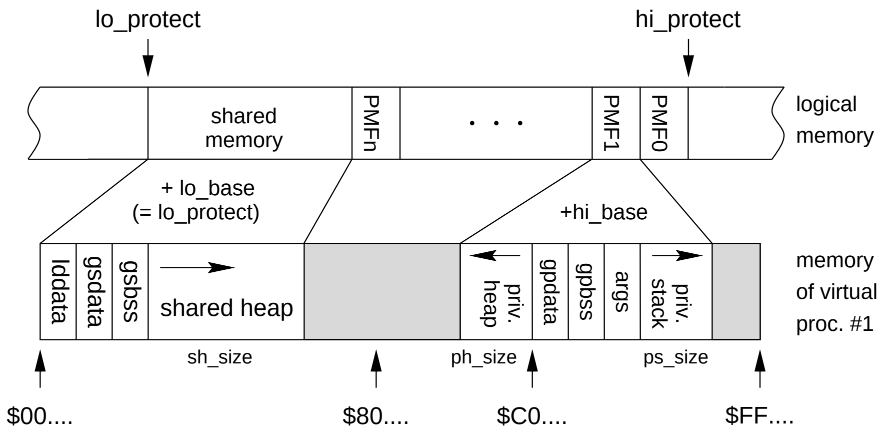

Figure 5: Memory layout

processors. Wire processors execute tasks from the wire queue and generate tasks for the route queues, route processors execute tasks from the route queues. To guarantee a good load balance, processors are allowed to switch queues if the queue they are currently accessing is empty. The multi prefix operations can be used to realize a sequentialization-free access without any degradation in the routing quality as observed in the original implementation. There, the lock was omitted for performance reasons, which had a negative impact on routing quality.

The radiosity program [26] is a simulation method from computer graphics to generate photo-realistic images with objects which diffusely reflect or emit light. The main computational effort consists in computing the *configuration factors* between different patches, which determine the light transfer between patches. The configuration factors only depend on the geometry of the environment. The hierarchical radiosity method adaptively subdivides the surfaces into a hierarchy of patches where the subdivision of each surface is represented by a quadtree. The final radiosities of all patches are specified in a linear system of equations describing all the interactions. The system is solved by a generalized Jacobi-iteration scheme. The iteration stops when the difference of the total radiosity of two successive iterations is small enough. In each iteration step the interactions between patches are reconsidered which may result in a change of the interaction levels or in new subdivisions. In the original SPLASH-2 implementation each processor has its own task queue and inserts new tasks created by local tasks into this queue to achieve locality. Task stealing realizes load balancing. Improvements of this parallel implementation include a parallel construction of the BSP-tree which stores the input polygons and allows for a fast visibility check, the use of a parallel task queue, and the use of parallel loops where locality can be ignored. Besides the improvements of the efficiency, our investigations show that the optimized version leads to a much simpler source code.

MP3D [29] is the particle simulator taken from SPLASH-1 for fluid dynamics problems. It uses a uniform subdivision in cubic cells. For an efficient computation of collisions, the active space is represented as a 3-dimensional array of space cells. The collisions are determined statistically using collision probabilities. Our parallel implementation exploits particle-based parallelism instead of a cell-based parallelism to obtain a fine-grained parallelism of high degree. Another advantage of the particle based view is that the amount of computations per particle does not vary as much as the work per space cell. The dynamic mapping of the tasks to the processors is performed by parallel accesses to a single task queue. The dynamic mapping guarantees an almost balanced execution of the tasks if the number of particles exceeds the number of processors. The achievable speedup is limited only by conflicts caused by concurrent accesses to space cells. The change of the static mapping of the original SPLASH implementation to a dynamic mapping leads to an increase of the relative speedup of 20 to 70 percent, depending on the number of particles.

In the conservative circuit simulation PTHOR from the SPLASH-1, the deadlock detection phase can be simply implemented by maintaining a shared counter, which is initially set to zero and subsequently accessed via multi prefix add instructions. A processor whose event list gets empty increases the counter. It decrements the counter if it finds a new event to simulate. A deadlock is detected if the counter equals the number of available processors. In the deadlock resolution phase, the minimum time stamp can be computed with multi prefix operations. The concept of NULL-messages which usually increases the amount of necessary communications can be used in the SB-PRAM as a means to speedup certain circuit simulations. The degree of parallelism is often quite low for circuit simulations, nevertheless, on the SB-PRAM many input patterns can be handled simultaneously where the circuit description is held only once in memory. For a more detailed analysis see [17].

Graphics. Graph layout and visualization of such a layout is an important issue in many interactive applications. We parallelized one method which realizes a fish-eye lens on a layouted graph where the focus is given by a polygon. The parallelization is performed with a parallel queue over all nodes of the graph. A processor calculates the displacement for a node taken from the queue by a scanline method independently from any other node and processor. The parallel queue allows for almost optimal load balancing without prior knowledge of the layout of the input graph.

We parallelized one of the fastest ray tracers [4]. The data base is held in shared memory only once, so that even large scenes can be rendered with the parallel machine. A parallel pixel queue containing the primary rays allows for almost optimal load balancing. The concept of light caches which hold the last objects causing shadows was adapted for the parallel case. All processors have access to the same cache structure. The run times on the 4-SB-PRAM matched exactly the run times predicted with the simulator. Because of the high degree of parallelism of this application almost linear speedup is achieved as long as the resolution of the image is not to small. In [3] it is shown that a 128-SB-PRAM will achieve seven times the performance of a one processor SGI challenge workstation.

Future applications. Besides the applications described so far, we have investigated possible commercial applications which profit from the high bandwidth to memory, from the large number of hard disks and from the fine grained parallelism with efficient queue data structures of the SB-PRAM.

With the help of time discrete event simulations where the behavior of the SB-PRAM was incorporated to a certain level we have been able to predict the performance of the machine for a transaction system. The base of the analysis was the DEBIT/CREDIT benchmark [6]. Disk I/O and instruction counts have been modeled in detail. Each transaction is entirely processed by one virtual processor. All transactions are held in a parallel queue. Because in the PRAM model no communication time has to be considered, classical performance prediction by code inspection was possible. Waiting times due to locks on data structures are obtained by an event simulation. The transaction throughput of a complete SB-PRAM will lie in the range of 6000 to 10000 transactions per second. The detailed analysis and a comparison with other machines is contained in [11].

Video-on-Demand servers provide a repository of movies stored on a disk array in compressed format, e.g., motion JPEG, MPEG or MPEG-II. When users connect to the server and request to watch a movie, the server must guarantee the uninterrupted play-out of a video sequence. The SB-PRAM already has two disk controllers on the local processor boards and an ATM interface for video output shall be connected to the extension bus. For this application, the virtual processors of a physical processor are divided in two halves: one half is responsible for accessing disks and moving data to global memory, the other half loads data from global memory and moves it to the ATM interface. If multiple users want to watch the same movie, main memory is used as a global cache for video streams. The control structures of this application can be implemented by parallel data structures without serialization. First simulations [10] have shown that the 128-SB-PRAM can support more than 1280 video streams with 600 KB/s each using parity enhanced random striping on 256 disks.

ATM technology is the designated standard for the worldwide, broadband ISDN network. The availability of high-bandwidth ATM switching nodes that can handle thousands of ATM connections is of crucial importance for this network. ATM relies on the concept of statistical multiplexing, i.e., the capacity of outgoing ATM links is not exceeded in the long term, but short time overload conditions may occur. Actually, the overload is avoided by buffers on the outgoing ATM links. Simulations show that several Megabytes may accumulate in a buffer, but only a few buffers in a switch are filled simultaneously. This imposes a tradeoff for the design of a switch: either one wastes resources with private buffers for each link or decreases performance with a single shared buffer. However, the SB-PRAM as a shared memory machine provides both the high bandwidth to memory and the routing functionality for the switching node. The routing tables describing the ATM connections may be held in shared memory, too. Details of an SB-PRAM as ATM switch are gathered in [8].

4. The 128-SB-PRAM

We now discuss hardware modifications for the planned machine with 128 physical processors and 8 GByte of shared memory. All boards will be redesigned to facilitate testing and debugging. Furthermore, network link technology will be changed and the processor board will be extended by some features.

We have experienced that debugging is rather complicated on a 4 processor machine. For the 128-SB-PRAM, we intend to design for testability employing JTAG boundary scan for checking the boards. This technique will be integrated into an automated monitoring and debugging software, so that malfunctioning boards can be easily singled out. The number of chips on the boards will be reduced by moving to newer chip technology. In this way, the design will get more compact and less sensitive to failures.

In the 4-SB-PRAM, we used ribbon cables for the network links, because they were cheap and sufficient for the 5 MHz clock speed. In the 128-SB-PRAM, we will switch to high speed differential serial lines. Besides better electrical properties, the cabling becomes less difficult, because the coaxial cable is more flexible and needs less space. The extension port of the local processor bus will be changed to PCI standard. This enables us to attach standard interface cards for ATM and graphics.

For the Video-on-Demand and ATM switch applications mentioned in Section 3.2, we also want to speed up the transfer of data between local bus and global memory. In the current design, the CPU needs two instructions to transfer data due to its load-store architecture. In spite of this, the performance can be doubled by modifying the data paths (cf. Figure 6).

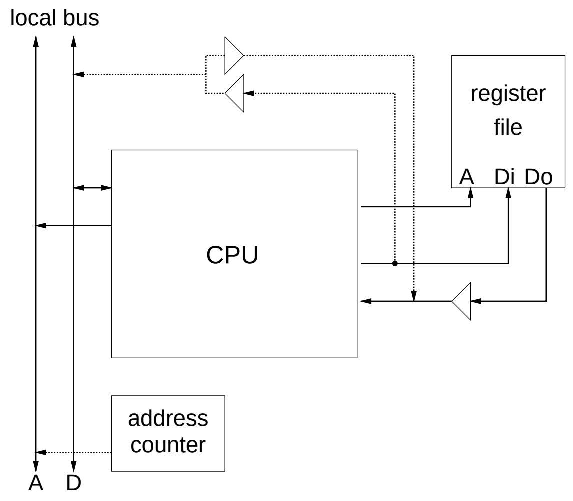

Figure 6: Data paths for the pseudo DMA

The basic idea is to redirect the access to a particular register *R*, called the *pseudo DMA register*, to the local address space. Instead of reading or writing the contents of *R* to the register file, the CPU reads or writes the contents of a local memory cell. For each access, the address *A* is increased by an adjustable value using an address generator. Before a transfer is initiated, *A* will be initialized to a proper start value. Using the pseudo DMA register, the CPU can transfer a data word between local bus and global memory in only one cycle.

5. Conclusion

The 4-SB-PRAM is a working prototype of a shared memory multiprocessor which emulates a priority CRCW PRAM. We described briefly some details of the design. The design methodology was guided by simultaneous development of hardware and software. Several decisions have been drawn towards a simple and less expensive direction. The programming environment includes two compilers with additional parallel libraries, an instruction level simulator, a basic operating system PRAMOS, and a controlling and monitoring tool.

We have implemented several applications from the SPLASH application suite and two complex applications from computer graphics. Furthermore, we analyzed the performance of the machine on applications that require large shared buffers (ATM switches), irregular access to shared memory (data base management), and high bandwidth to the file system (Video-on-Demand). The performance achieved on these application must be analyzed more thoroughly and compared to other parallel machines.

The current research is directed towards the construction of a 128-SB-PRAM. The chip set has been produced already. We outlined the necessary modifications of the remaining hardware. A parallel shared file system is under development.

Acknowledgments

We would like to mention additional work done within the project that cannot be reflected sufficiently in the references. Up to know there have been 24 master's theses, four PhD theses and several non-academic works. Interested people are referred to `http://www-wjp.cs.uni-sb.de/SBPRAM/`.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and Dieter Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. of the 1990 International Conf. on Supercomputing*, pages 1–6. ACM, 1990.

[3] A. Formella. Ray Tracing Complex Scenes: Parallel or Sequential? In *Proc. of* 7^{th} *IASTED/ISMM International Conf. on Parallel and Distributed Computing and Systems*, pages 89–92. IASTED–Acta Press, October 1995.

[4] A. Formella and C. Gill. Ray Tracing: A Quantitative Analysis and a New Practical Algorithm. *The Visual Computer*, 11(9):465–476, December 1995.

[5] R.M. Butler and E.L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[6] Transaction Processing Performance Council. TPC Benchmark^{TM} B – Revision 1.1. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing (2nd edition)*, pages 87–129. Morgan Kaufmann Publishers, 1993.

[7] D. Cross, R. Drefenstedt, and J. Keller. Reduction of network cost and wiring in Ranade's butterfly routing. *Information Processing Letters*, 45(2):63–67, 1993.

[8] R. Drefenstedt, J. Keller, and W.J. Paul. Applications of PRAMs in Telecommunications. In *Proc. of the 13th World Computer Congress, IFIP Congress '94*, vol. 1, pages 203–210. Elsevier Science Publ. B.V., 1994.

[9] 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 Computer Science 694, pages 1–11. Springer, June 1993.

[10] J. Friedrich, T. Grün, and J. Keller. Video-on-Demand on the SB-PRAM. In *Proc. of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 105–111, April 1996.

[11] C. Gemünd, M. Jakob, W. Massonne, W.J. Paul, and B. Spengler. High performance transaction systems on the SB-PRAM. In *Proc. of the 3rd Isreal Symposium on the Theory of Computing and Systems*, pages 1–10, 1995.

[12] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C–32(2):175–189, February 1983.

[13] T. Grün, T. Rauber, and J. Röhrig. The Programming Environment of the SB-PRAM. In *Proc. of 7th IASTED/ISMM International Conf. on Parallel and Distributed Computing and Systems*, pages 504–509. Acta Press, October 1995.

[14] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A high–level–language for PRAMs. *Future Generation Computer Systems*, 8(4):379–393, September 1992.

[15] T.J. Harris. A Survey of PRAM Simulation Techniques. *ACM Computing Surveys*, 26(2):187–206, June 1994.

[16] R.M. Karp and V.L. Ramachandran. A survey of parallel algorithms for shared–memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. A*, pages 869–941. Elsevier, 1990.

[17] J. Keller, Th. Rauber, and B. Rederlechner. Conservative Circuit Simulation on Shared–Memory Multiprocessors . In *Proc. 10th Workshop on Parallel and Distributed Simulation*, pages 126–134, Philadelphia, USA, May 1996.

[18] J. Keller. Fast rehashing in PRAM emulations. In *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 626–632, December 1993.

[19] J. Keller, W.J. Paul, and D. Scheerer. Realization of PRAMs: Processor design. In *Proc. WDAG '94, 8th International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 857, pages 17–27. Springer, September 1994.

[20] J. Keller and Thomas Walle. A note on implementing combining networks. *Information Processing Letters*, 55(4):195–200, August 1995.

[21] C.W. Kessler and H. Seidl. Fork95 Language and Compiler for the SB-PRAM. In *Proc. of the 5th Int. Workshop on Compilers for Parallel Computers*, pages 408–420, 1995.

[22] C.W. Kessler and H. Seidl. Integrating Synchronous and Asynchronous Paradigms: the Fork95 Parallel Language. In *Proc. of MPPM-95 Conf. on Massively Parallel Programming Models*, pages 134–141, 1995.

[23] C.W. Kessler and J.L. Träff. A library of basic PRAM algorithms and its implementation in FORK. In *Proc. 8th Annual Symposium on Parallel Algorithms and Architechtures (SPAA)*, page forthcoming, 1996.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[25] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. of the 1985 International Conf. on Parallel Processing*, pages 764–771. IEEE, 1985.

[26] A. Podehl, G. Rünger, and T. Rauber. Scalability and Granularity Issues of the Hierarchical Radiosity Method. In *Proc. of Euro-Par' 96*, vol. 1, pages 789–798. Springer LNCS, August 1996.

[27] A.G. Ranade. How to emulate shared memory. In *Proc. of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[28] A.G. Ranade, S.N. Bhatt, and S.L. Johnson. The Fluent Abstract Machine. In *Proc. of the 5th MIT Conf. on Advanced Research in VLSI*, pages 71–93, Cambridge, MA, 1988. MIT Press.

[29] T. Rauber, G. Rünger, and C. Scholtes. Shared-memory Implementation of an Irregular Particle Simulation Method . In *Proc. of Euro-Par' 96*, vol. 1, pages 822–827. Springer LNCS, August 1996.

[30] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[31] B.J. Smith. A pipelined shared resource MIMD computer. In *Proc. of the 1978 International Conf. on Parallel Processing*, pages 6–8. IEEE, 1978.

[32] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. A*, pages 943–971. Elsevier Science Publishers and MIT Press, 1990.

[33] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.