FAULT-TOLERANCE MECHANISMS IN THE SB-PRAM MULTIPROCESSOR

MICHAEL BRAUN[1] ANDREAS GRÄVINGHOFF[2] JÖRG KELLER[2]
[1]Universität des Saarlandes, Computer Science Dept., 66041 Saarbrücken, Germany
[2]FernUniversität-GHS Hagen, Computer Science Dept., 58084 Hagen, Germany

ABSTRACT

The SB-PRAM is an experimental multiprocessor architecture with a shared address space and synchronously running threads, i.e. giving the illusion to work on a PRAM. A 4-processor prototype has been completed while a 64-processor prototype is under construction. We investigate the detection and handling of single bit errors occuring during transmission of packets in the interconnection network. We analyze the impact of an error on the different parts of a packet and derive several strategies to recover from such an error. The strategies range from single bit correction codes to checkpointing the application and roll back in case of error. We find that the changes necessary in hard- and system software are small. In particular, none of the ASICs designed for the SB-PRAM have to be changed. The runtime overhead due to the fault-tolerance mechanisms can be neglected. Finally, we sketch how these strategies can be extended to cover component failures.
Keywords: Fault-Tolerance, Interconnection network, Transient errors, Checkpointing, Recovery strategy.

1 INTRODUCTION

Many investigations on the fault-tolerance of multiprocessors focus on the detection and the handling of component failures. Our case study investigates the consequences and the handling of single bit errors occuring in the interconnection network during transmission between routing chips. The machine under construction confines a large number of boards and cables into a small physical space (three 19-inch racks). We consider the off-board network links to be most susceptible to the resulting electromagnetic field. Therefore we focus on single-bit errors (e.g. caused by crosstalk) on off-board links. While one may argue that these transient errors can be handled by applying an error-correcting code on the transmitted information, we try to derive a cheaper solution by a detailed analysis of the consequences of errors on the different parts of that information.

The remainder of the paper is organized as follows. In Section 2 we describe the SB-PRAM, an experimental multiprocessor. We will focus on the description of the interconnection network and routing protocol. In Section 3 we analyze the effects of a single bit error depending in which part of the transmitted information it occurs. We describe several strategies to correct such an error, which include applying an error-correcting code on part of the transmitted information, and checkpointing. Section 4 derives the modifications of hardware and software necessary to implement the strategies just developed. In Section 5 we summarize our results and give an outlook how the proposed strategies can be extended to cover component failures, thus amortizing the cost of the necessary modifications.

2 SB-PRAM ARCHITECTURE

The SB-PRAM [1] is an experimental multiprocessor developed and prototyped at Universität des Saarlandes in Saarbrücken, Germany. It provides a shared address space while memory is physically distributed. In contrast to machines such as SGI Origin [2], all user threads synchronously execute assembler instructions and access time to the shared address space is uniform. Thus, the SB-PRAM gives the user the illusion to work on a PRAM [3], which is a popular model to design and analyze parallel algorithms. One of the advantages of the PRAM model is a simpler algorithm analysis leading to better performance prediction and less time spent with algorithm tuning.

To implement the uniform memory access time, a randomized mapping (more exactly, universal hashing) from address space to memory modules is used. This ensures (with high probability) an even distribution of all accesses in one step. A processor P_i wishing to access a shared address sends a request via a *request network* to the appropriate memory module M_j , in case of a `load` instruction the content of the address is returned via a *return network*. As both networks have some latency, the access time is uniform but long. The latency is hidden from the user by using a multithreaded processor. So far, the SB-PRAM architecture is similar to the Tera MTA [4]. However, in contrast to that machine, context is switched after each instruction of every thread to realize the synchronous execution mentioned above. For the same reason, accesses from different steps of the machine are separated within the networks.

On applications with irregular access patterns (see e.g. [5]) the uniform memory access time gives the SB-PRAM advantage over machines which try to avoid remote access by using coherent caches. On the latter machines, i.e. SGI Origin, much tuning time must be spent to achieve locality in order to exploit the caches. The SB-PRAM alleviates the programmer from doing these things, which often leads to simpler algorithms for irregular applications.

Both the request network and the return network are unidirectional butterfly networks. The paths from any input to any output are unique in these networks. Path selection

in the request network can be done by the routing switches based on one bit of the address. Each routing switch selects the request/reply with the smaller address first. After executing one step from each thread, a processor inserts an end-of-round (`EOR`) message into the network. These `EOR` messages are used to separate memory requests from different steps.

The above routing algorithm ensures that the order of replies passing a switch in the return network is the same as the order of the requests for these replies passing through a switch of the request network. Therefore, no routing decision is needed in the return network. The routing decisions of each switch in the request network are simply stored in a FIFO queue called *direction queue* , and recalled in the return network. Hence, corresponding switches of both networks are implemented in one chip.

There are 8 different access modes, which can be partitioned into classes `load` and `store`. `Load` requests carry only the type (3 bits) and an address (32 bits), `store` requests carry the type, an address and a 32 bit data word to be stored. Requests of type `store` are transmitted in two flits between routing switches. The first one carries type and address, the second one carries the data word. Requests of type `load` are transmitted in one flit. In the return network, only data words are transmitted in two flits of 16 bits each. The flits are not marked "first" or "second", they are recognized in a switch by a simple finite state automaton based on the rules above.

In both interconnection networks, the control flow is realized via *valid* and *busy* signals. The sending routing switch indicates transmission of a flit via the *valid* signal. The receiving routing switch uses the *busy* signal to indicate that no more flits can be handled (i.e. the input queue is full). Upon receiving such a *busy* signal, the sender suspends transmission of flits.

A 4-processor prototype has been completed [6] and assembly of a 64-processor prototype is currently under way. The routing switch used in the network is implemented as a gate array. No fault-tolerance was considered during the design except parity generation/checking. Therefore, our goal is to introduce fault-tolerance mechanisms to the SB-PRAM without any changes to the gate array and without compromising performance.

3 ERROR ANALYSIS

The fault model used to determine the effect of errors on the routing network are single bit errors occuring during transmission between routing switches. Since the SB-PRAM prototype confines a large number of boards and cables into a small physical space, such errors may be caused by crosstalk.

The effect of such errors depends on the affected part of a request or return message: If the data word of a request is affected, a wrong value will be written. Likewise, a wrong value will be returned, if the data word of a return is affected. A faulty address part will cause the wrong cell to be accessed. Since routing information is derived from the address, the request may even be routed to the wrong memory module depending on the affected address bit. These errors can compromise the application, but not the routing algorithm.

Due to erroneous mode information, a `load` request can be transformed into a `store` request, or vice versa. In these cases, the direction queue will not be filled or will be filled erroneously, respectively. In addition, if the number of flits per request differs between old and new modes, the next flit(s) will be treated wrongly. This stems from the fact that the flit type is not derived from the flit itself, but from the finite state automaton mentioned in the previous paragraph. By this second set of errors, the routing algorithm will be compromised. For example, an erroneously introduced `load` request leads to a reply that cannot be handled by those switches where the request was still intact, i.e. of type `store`.

Errors on the control signals can be fatal as well: A lost busy signal will lead to lost flit(s), if more requests are sent than the receiving switch can handle. If a flit that belongs to a multiple-flit (i.e. `store`) message is lost, the next flit will be treated wrongly. An extra busy signal causes the sending routing switch to suspend transmission, which only causes a delay and can thus be tolerated. If a valid signal is lost, the corresponding flit is not recognized at all and thus lost. Again, if the flit belongs to a multiple-flit request, the following flit(s) will be treated wrongly. An extra valid signal introduces a "trash flit". This flit will cause a false access. Depending on the mode, the direction queue may be filled erroneously and the next flit(s) may be treated wrongly.

Since errors on mode, busy and valid are compromising the routing algorithm, such errors must be not only detected, but corrected as well. Errors on the address and data words of a message only need to be detected, if computation can be rewound to eliminate false accesses.

Based on these observations above, we derive the following strategy to handle single-bit errors: The routing switch ASIC already supports error detection via parity detection and generation on all incoming and outgoing links, respectively. Error correction is applied by external hardware on the critical parts (mode, busy, valid) of a message only. Non-critical errors on requests or replies are detected at a memory module or a processor, respectively. Upon detection of an error, the corresponding processor/memory module informs the host about this event via the host bus. The host immediately freezes the machine by stopping all processors simultaneously. The network may still contain valid messages that need to be flushed. Rather than waiting for the network to settle, a network reset is performed. This reset initializes all routing switches and clears all queues in the network. At this point, processor and memory states are fixed, since there are no more requests/replies that may change the states. Therefore, processor and memory states can now be rolled back to the last checkpoint. Restoration of memory state involves applying all writes to memory since the last checkpoint (recorded in a queue) in reverse order. This operation is done by special hardware. After memory state has been restored, restoration of processor state is done by loading the entire register set with values

stored in dedicated buffers at the last checkpoint. Restoration as well as saving are performed by a combination of hard- and software, which is described in detail in the next section. In the cause of this operation, the host unfreezes all processors. After the processors exit the global exception, processor state is restored and computation continues from the last checkpoint.

4 MODIFICATIONS

Implementation of the strategy derived above requires a moderate amount of additional hardware at the processor, network and memory nodes. For all of these components, we will discuss this hardware as well as consequences on performance in detail.

4.1 PROCESSOR MODULE

The state of each thread consists of 30 universal registers and five special registers, namely the status register `SR`, the program counter `PC`, the base and protection registers `BASE`, `PROT` and the load request register `LD`. Saving the processor state involves saving these 35 registers for each thread. The universal registers are implemented in an external memory due to the limited number of available gates on the processor gate array. This external memory allows saving and restoring of processor state by hardware. However, the special registers are implemented on-chip. Therefore, we use a combination of hard- and software to save/restore the processor state.

A global exception is generated by external hardware using an already present instruction counter. Starting with the first thread, all threads will enter the corresponding exception routine shown below: Instead of storing the five special registers to the specified universal registers, accesses to the external register memory are redirected to a backup memory during execution of the exception by special hardware. After execution of the global exception has been completed, the processor is halted. Now the content of the external register memory is copied into the backup memory.

The SB-PRAM processor accesses the external register memory via several busses for adresses of odd (`AO`) and even (`AE`) threads and incoming (`DIN`) and outgoing (`DOUT`) data. The external memory was split in two parts in order to faciliate two simultaneous operations per clock cycle. During normal operation the backup memory is isolated and the additional control logic keeps track on the number of completed rounds and initiates the global exception. During execution of the global exception the register memory is isolated and all accesses are redirected to the backup memory at adresses generated by the control logic. Later, when the processor has been halted and the universal registers are saved to backup memory, the processor address and data busses are isolated. Register and backup memories are accessed on control-logic generated addresses.

Processor state is saved every x instructions; the last two states are kept in the backup memory. The value of x

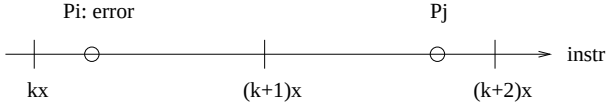

FIG. 1: CHECKPOINTING INTERVAL

depends on the routing network: As is illustrated in Figure 1, processors may be in different rounds, which can be explained as follows: A processor that performs only local computation, will insert only `EOR` messages into the network and never wait. Other processors may have to wait for outstanding `load` requests, if the latency can not be completely hidden because of hot spots in the network. When an error occurs in a round between kx and $(k+1)x$, x must be large enough to ensure that no processor is past the point $(k+2)x$, since then the processor state for round kx is no longer available.

An upper bound on x can be constructed in the following way: As was said above, every processor inserts an `EOR` message into the network after each round. The maximum drift between any two processors is then determined by the maximum number of `EOR` messages that can be stored on an arbitrary path through the network. Hence, x can be obtained by adding the queue sizes along such a path, which yields $x \leq 256$ for the 64-processor prototype.

The register memory can perform two accesses every clock cycle, thus $30 \cdot 32 \cdot 1/2 = 480$ cycles are required to save or restore the universal registers of all threads. Since an instruction requires four clock cycles and there are 32 threads, saving or restoring the universal registers requires $480 \cdot 1/4 \cdot 1/32 = 3.75$ instructions per thread. Along with the seven instructions of the global exception, saving or restoring the processor state requires 10.75 instructions per thread. This translates to a maximum performance loss of

$$\frac{256 + 10.75}{256} = 4.03\%$$

per thread, if no errors are encountered. In the case of errors, every restore operation will cause additional overhead of 10.75 instructions per thread.

4.2 NETWORK NODES

To ensure correction of single bit errors on the critical components of a message (mode, busy, valid), hamming codes are used. Corresponding hardware is inserted between the routing switch ASICs and the network. Note that the incoming flits enter the routing switch on two consecutive half-cycles due to pin restriction in the routing switch gate array. Commercial EDC circuits (e.g. IDT 49C465) can detect and correct errors using hamming codes in less than 15 ns on 32 bit words. However, the timing of the incoming and outgoing links is tight and does not allow for such an additional delay. Even in our case, where there are only a small number (5) of signals to be protected by hamming codes, there might not be enough time available, since we implement the EDC circuits via programmable logic. Therefore we introduce an additional pipeline stage at the

incoming and outgoing links, which allows us to utilize almost a full clock cycle for EDC. This additional pipeline stage has to be taken in account when generating busy signals. As the routing switch can be configured to generate a busy signal at an arbitrary number of queue entries, this poses no problem. One may argue that now even error detection and correction on whole flits is possible, but the following reasoning shows that the associated cost is not acceptible: Since the network consists of five stages of printed circuit boards (processor, memory and 3 network stages), there are 4 (number of links between stages) times 64 (number of processors) times 2 (EDC required at both ends) times 2 (number of chips per link assuming 32 bit data bus width) = 1024 additional EDC circuits required. Note that this calculations assumed that request and return links use the same chips, which may pose technical problems due to different timing requirements. The larger number of chips will have an significant impact on the cost, power and board area consumption.

4.3 MEMORY MODULE

To allow for the restoration of memory state, all write accesses to a memory module must be logged along with the overwritten values. To restore the old memory state, all logged transactions are rolled back in reverse order. We use a queue of size q to log every write access, which must be large enough to handle all write accesses to a single module during an interval of $2x$ rounds. In the worst-case, every thread writes to the same module for a duration of $2x$ rounds. This yields a maximum queue size of $q \leq 2 \cdot x \cdot t$, where t is the number of threads in the whole machine. A 64-processor machine will therefore need a queue of $2 \cdot 256 \cdot 2048 = 1024\,\mathrm{K}$ entries on every memory module. In practice, a much smaller queue size will suffice. First of all, it is very unlikely for every thread to perform memory writes for $2x = 512$ rounds in a row. Second, accesses to the same memory module are unlikely as well, since addresses are hashed (i.e. a mapping is used in order to achieve a similar load on all memory modules). In addition, if the number of processor wait cycles (e.g. caused by memory congestion) reaches a certain threshold, *rehashing* (i.e. switching to a different mapping) is performed routinely in the SB-PRAM [7].

Nevertheless, should an overflow occur, this fact is reported to the host. As already outlined above, the hosts subsequently restores processor and memory states to the last checkpoint. In the case of an overflown queue, rehashing is perfomed in addition before computation resumes.

5 CONCLUSIONS

An investigation of the SB-PRAM on fault-tolerance was performed using a single-bit error model. A detailed analysis of possible errors was used to design a recovery strategy to handle such errors. The presented strategy can be implemented with no modifications to the routing switches and processor chips, as all additional hardware can be put off-chip. In the case of the processor node, the additional logic can be implemented using programmable logic (two CPLDs), a memory chip and standard logic. The EDC circuits at the network nodes can be implemented with programmable logic, since the number of signals to be protected is very small. The on-chip implementation of error correcting codes on whole flits would have increased the critical path in the routing switch. For example, the parity generation logic is on the critical part for the routing switch [8]. At the memory nodes, the queue control logic can be added to the already present FPGA. This may require a larger FPGA with more gates and pins. Since the FPGA used at present is quite small (13.000 gates according to Xilinx), this poses no problem. In all three cases, a redesign of the printed circuit board will be necessary. However, as the changes are located in small, closed sections of the different boards, this is neither difficult nor expensive. We plan to simulate memory system behaviour in order to obtain information about practical queue sizes.

Failure of a routing switch can be handled by using a modified butterfly network, where paths are not unique. This can be achieved by placing a copy of the last network stage before the first and the ability to bypass faulty routing switches on the first and last stages. The strategy to recover from the failure remains largely identical. Additionally the network must be reconfigured to allow bypassing the faulty switch. Failure of processors can be handled by checkpointing and migration of threads. To do this, a small number of threads is reserved at every processor and processor states are saved to disk at regular intervals (larger than x). Upon failure of a processor, the corresponding threads are loaded from disk and distributed among the reserved threads of the remaining processors. A mechanism to handle failure of memory modules in the case of randomized (i.e. hashed) addressing has been presented by Savva and Nanya [9].

REFERENCES

[1] F. Abolhassan et.al. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, 1993.

[2] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. 24th Annual Int.l Symp. on Computer Architecture*, pages 241–251, 1997.

[3] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Annual Symp. on Theory of Computing*, pages 114–118, 1978.

[4] R. Alverson et.al. The Tera computer system. In *Proc. 1990 Int.l Conf. on Supercomputing*, pages 1–6. ACM, 1990.

[5] Arno Formella et.al. Scientific Applications on the SB-PRAM. In *Proc. Int.l Conf. on Multi-Scale Phenomena and Their Simulation*, 1997.

[6] P. Bach et.al. Building the 4-processor SB-PRAM prototype. In *Proc. 30th Int.l Symp. on System Science*, volume 5, pages 14–23, 1997.

[7] J. Keller. Fast rehashing in PRAM emulations. *Theoretical Computer Science A*, 155:349–363, 1996.

[8] T. Walle. *Das Netzwerk der SB-PRAM*. Dissertation, Universität des Saarlandes, 1997.

[9] A. Savva and T. Nanya. Gracefully degrading systems using the bulk-synchronous parallel model with randomised shared memory. In *Proc. 25th Int.l Symp. on Fault-Tolerant Computing*, pages 299–308, 1995.